21/8/2012

No wait :

It is an optional clause, used along with for update clause, this clause returns control to the current session, if another user is locking the resource also.

In this case oracle server throws

ora-54 : Resource busy.

U1 | U1

SQL> select * from scott.emp
where deptno = 10 for update;

Scott | tiger

SQL> select * from emp (when no wait is not used, it does not allow to perform other operations as it gets stuck)
where deptno = 10
for update no wait;

ora 54: Resource busy

SQL> —

Table level locks :-

In this method we lock entire table, there are 2 types of table level locks used by developers

1) Share lock
2) Exclusive lock.

Share lock : when we are using this lock another user query the data and they cannot perform dml operations and also at a time number of users lock the resource

Syntax :- lock table tablename in share mode;

U1 / U1

Sql> select * from scott.emp;

sql> lock table scott.emp in share mode; (✓)

scott/tiger

sql> lock table emp in share mode; ↵

Exclusive mode:

when we are using this mode another user query the data but they cannot perform dml operations.

Also at a time only one user can lock the resource

Note: whenever we are using cursor locking mechanism internally oracle server use exclusive mode.

Syntax :- lock table tablename in exclusive mode;

sql> lock table table emp in exclusive mode;

U1/U1

sql> select * from scott.emp;

sql> lock table scott.emp in share mode; ↵ (✗)

scott / tiger

sql> lock table emp in exclusive mode; (✓)

Set operators:

These operators are used to retrieve data from single or multiple tables.

These operators are also called as vertical joins.

1.) union → returns unique values
2.) union all → unique + duplicate values
3.) intersect → Common values.
4.) minus → values in 1st query are not in 2nd query.

```
select job from emp where deptno = 10
union
select job from emp where deptno = 20;
/
```

Note: when we are using set operators always corresponding expressions must belong to same data type

set operators always returns first query column names as column headings

Ex: select ename from emp
union
select dname from dept;
/

o/p: Ename

→ If Corresponding expressions doesnot belongs to same data type also we can retrieve data from multiple queries, in this case we use appropriate type Conversion functions

Ex:-

```
Select ename "name", to_number(null) "deptno"
    from emp
    union
    select deptno, to_char(null), deptno from dept;
```

Ex:-

Ename	dname
...	...

Conversions:-

→ Converting one data type to another is called Conversion.

There are 2 types of Conversions supported by Oracle

1.) Implicit Conversions
(or)
automatic Conversions

2.) Explicit Conversions

Implicit Conversions:- whenever expressions contains character data oracle server internally checks that character type having pure number or not

If it is a number oracle server automatically uses implicit Conversion.

Ex:

sql> select sal + '100' from emp;

Note: In oracle when we try to convert string type to number type oracle server returns an error as:

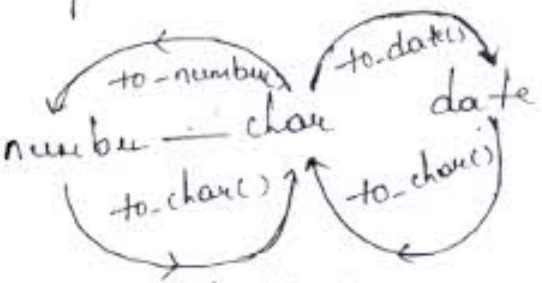
ora-1722 : invalid ~~char~~ number

Ex: sql> select sal + '100z' from emp;

error: invalid number

whenever we are using comparison operator (=) oracle server implicitly uses implicit conversion to convert date string into to date type.

sql> select * from emp where hiredate = '23-JAN-82';



number $\xrightarrow{\text{to\_char()}}$ char

char $\xrightarrow{\text{to\_number()}}$ number

char $\xrightarrow{\text{to\_date()}}$ date

date $\xrightarrow{\text{to\_char()}}$ char

To_number(): this conversion functn converts a string representing with format into without format

Ex: sql> select to_number('$45.6', '$99.9') + 3
from dual;

DP 48.6

22/8/2012

to_char():

This is an overloaded functn which is used to convert number type into character type & also used to convert data type into date string.

Ex: SQL> select
to_char('123456.78', '$999999.99')
from dual;

Result $123,456.78

SQL> select to_char(sysdate, 'dd/mm/yyyy')
from dual;

o/p: 22/8/2012.

To_date(): It is used to convert date string into date type.

SQL> select to_date('16-JUN-05') + 5 from dual;

21-JUN-05

Nested table: PL/SQL Concept:

→ Nested table is a user defined type which is used to store multiple datatypes into single unit.
→ Table within another table is also called as nested table
→ Nested tables are introduced in Oracle 8.0
→ Generally if we want to store multiple data items we use index by tables in pl/sql

→ But we are not allowed to store these tables permanently in database

→ To overcome this problem oracle 8.0 introduced nested tables, Varrays, to store permanently in to database using sql

→ If we want to create user defined datatypes then we are allowed to create but by using some specific construction.

→ In C ... to create user defined datatype we use

typedef () {struct st

{

}

C++ ---> Class

→ In oracle we can also create our own user defined type using type kw

→ In sql these types are objects, nested tables, Varrays.

Object:

This is a user defined type used to store different datatypes into single unit.

It is also same as structures in 'C' lang.

Syntax: create or replace type typename

as object (attr1 datatype (size), attr2

datatype (size) ...)

/

Creating nested table

step 1 : Create an object

step 2 : Create a nested table type using object

Syntax :

Create or replace type typename as
table of object type

step 3 : Create original table.

syn : Create table tablename (col1 datatype(size), ... col2 datatype(size), coln nested table type) nested table coln store as

Create table tablename (col1 datatype (size), col2 datatype (size) coln nested table type) nested table coln store as any name;

Ex :

sql > Create or replace type k10 as object (bookno number(10), bookname varchar2(10), price number(10));

sql > Create or replace type phone as table of k10;

sql > Create table stud (sno number(10), sname varchar2(10), col3 phone) nested table col3 store as ssss;

sql > desc stud;

Result : sno number(10)
sname varchar2(10)
col3 phone

If we want to initialize data in pl/sql we must use Constructor name.

→ If we want to store data using data we must use Constructor name.

→ Here Constructor name is same as type name.

```
sql > insert into stud
      values (1, 'murali', phone (k10 (101, 'java' 300)
                                  k10 (102, 'plsql', 201)));

sql> select * from stud;
```

→ These values are permanently in database in sql

→ But in plsql this data will be stored temporarily

Partitions

→ These tables are created by DBA to improve performance of the application

→ These tables are created on very large db

→ Partition tables are used in backup & recovery process

→ There are 3 types of partitions supported by oracle

1.) Range - partition
2.) List - partition
3.) hash - partition

→ Partition tables are created based on partition key.

Range partition

partitions are created based on range of values

Syntax:
create table tablname (col1 datatype(size) ... ...)
partition by range (key colname)
(partition partitionname values less than (value) ...
(partition) partitionname values less than (max value))

→ to view particular partition
syn: select * from tablename partition (partitionname);

Create partition table:

sql> create table bb (sno number(10), sal number(10))
partition by range (sal)
(partition p1 values less than (1000),
partition p2 values less than (2000),
partition p3 values less than (3000));
sql> select * from bb partition (p2);

→ it stores only less than values but not the equal values also

List partition: introduced in oracle 9i (internet)
→ using list partitions, partitions are created based on character datatype column.

syn: create table tablname (col1 datatype(size) ...)
partition by list (key col)
(partition partitionname values (value1, value2) ...);

sql> create table bbb (id number(10), country varchar2(10))
partition by list (country)
(partition p1 values ('India', 'Pakistan'),
partition p2 values ('uk', 'canada')
partition other values (default));
sql> select * from bbb partition (other)

23/8/2012

~~PL/SQL~~

In this partition, we are partitioning the tables based on hash algorithm

Syntax: create table tablename (col1 datatype (size) ....)
partition by hash (key columnname)
partitions anynumber;

sql> Create table g10 (empno number(10), sal number(10))
partition by hash (empno)
partitions 5;

/

desc user_tab_partitions;

all partitions information stored under user_tab_partitions

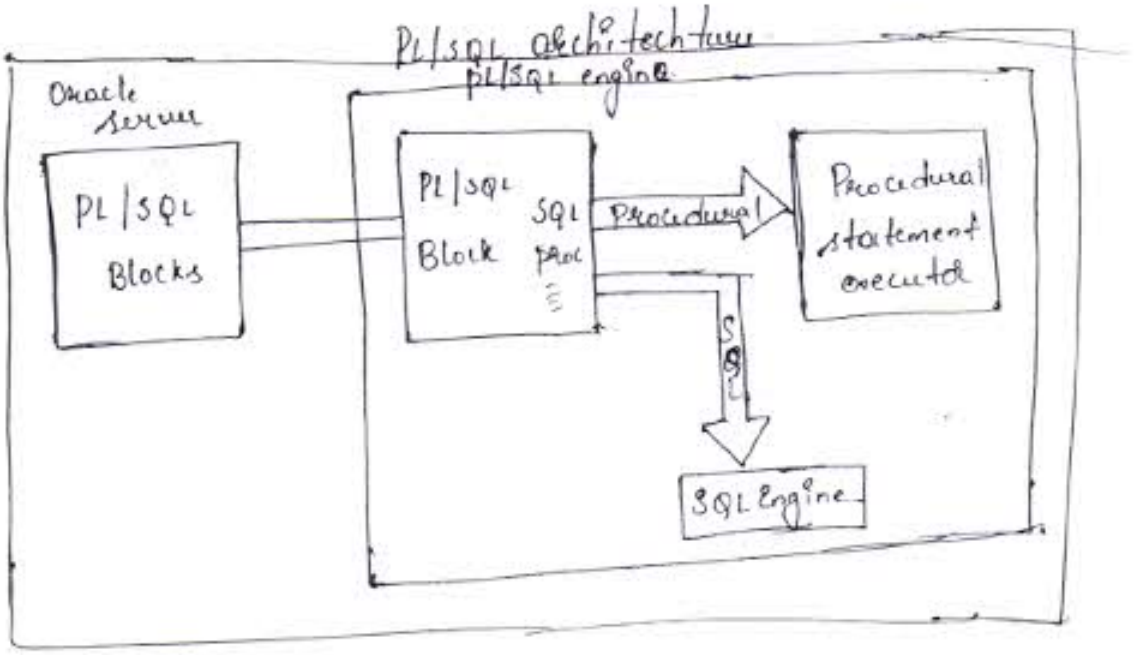
sql> select PARTITION_NAME from
user_tab_partitions where
TABLE_NAME = 'G10';

PL/SQL

→ PL/SQL is procedural language extention for SQL

→ It is a procedural language extention for SQL, it is the combination of procedural language, data manipulation language.

→ PL/SQL is a block structured programming language



Block structure:

Declare (optional)
→ variable declarations, cursors, user defined exceptions

Begin (mandatory)
→ DML, TCL
→ select ... into ...
→ if loops

Exception (optional)
→ handling exceptions;

end; (mandatory)

There are two types of blocks supported by PL/SQL

1) Anonymous blocks
2) Named blocks

Anonymous blocks
(No name given to these blocks)
(we are not allow to call these blocks explicitly)

Named blocks
(any name given to these blocks)
(we are calling these blocks explicitly)

Ex: Procedures, functions, triggers, packages.

Ex of anonymous blocks

```
Declare
==
Begin
==
end;
```

Declaring variables:-

Syntax: Variable name data type (size);

Ex:
```
declare
    a number (10);
    b varchar2(10);
```

a

Storing a value into variable
using assignment operator (:=)

Syntax: Variable name := value;

Ex:
```
declare
    a number (10);
begin
a:=10;
end;
```

To display value from variable or display a message

```
dbms_output.put_line ('message');
```

dbms_output → package name
put_line → procedure name

(or)

```
dbms_output.put_line (variable name);
```

SQL plus:

username : scott
password : tiger

As soon as we start sql plus we should set the environment variables as

```
sql> set serveroutput on;
sql> begin
       dbms_output.put_line ('welcome');
       end;
       /
```

Result: welcome

* to comment any line we use '--'

```
declare
  a number(10)
  a:=70;
  dbms_output.put line (a);
  end;
  /
```

24/8/2012

→ whenever we are using Not Null, Constant clauses we must assign the values at the time of declaring variables only.

```
declare
  a number(10) not null := 40;
  b constant number(10) := 5;
begin
  dbms_output.put_line(a);
  dbms_output.put_line(b);
end;
/
```

→ we can also use default keyword in place of assignment operator in declare section of the variables

```
declare
  a number(10) default 40;
begin
  dbms_output.put_line(a);
end;
/
```

Select ..... into clause

→ This clause is used to retrieve data from table into pl/sql variables, but this clause always returns single record or single value at a time

Syntax: select col1, col2..... into var1, var2 ...
from tablename
where condition;

↔ This stat. is used in executable section of the pl/sql block

write a pl/sql program for user entered emp no display name & sal of the employee from emp table

```
declare
v_ename varchar (10);
v_sal   number (10);

begin
dbms_output.put_line;
select ename, sal into v_ename, v_sal
from emp where empno = &no;
dbms_output.put_line (v_ename || '  ' || v_sal);

end;
/
```

o/p : Enter value for no : 7566.

JONES 1274.

write a pl/sql program to retrieve max salary from emp table and display that salary.

```
declare
v_sal number(10);

begin
select max(sal) into v_sal
from emp;
dbms_output.put_line (v_sal);
end;
```

```
declare
v_sal  number(10);
begin
select sal into v_sal
from emp where sal= max(sal);
dbms_output.put_line(v_sal);
end;
/
```

Here where condition is not used as group-functions do not allow where clause.

Note : We are not allowed to use group functions, decode Conversion function in pl/sql expressions

```
declare
a  number(10);
b  number(10);
c  number(10);
begin
   a:=40;
   b:=10;
   c:=greatest(a,b);  ⟹ Here max(a,b) should not be used.
dbms_output.put_line(c);
end;
/
```

26/8/2012

Variable attributes.

There are two types of variable attributes supported by pl/sql

1) Column level attribute
2) Row level attributes.

These attributes are used in place of data types in variable declaration or in place of data types in parameter declaration

When we are using these attributes pl/sql runtime engine automatically allocates memory based on corresponding column data types in a table.

Column level attributes:

→ In this method we are defining attributes (v) for individual columns.

In this case we use %type attribute

Syntax:- variablename tablename.columnname %type

Ex:
```
declare
v_ename emp.ename %type;
v_sal   emp.sal %type;
v_hiredate emp.hiredate %type;
begin
select ename, sal, hiredate into v_ename, v_sal,
v=hiredate from emp
where empno: &no,
```

```
dbms_output.put_line (v_ename || ' ' v_sal || ' '
                      v_hiredate);
end;
/
```

Row level attributes :-

In this method a single variable represent all different data types, an entire row in a table.

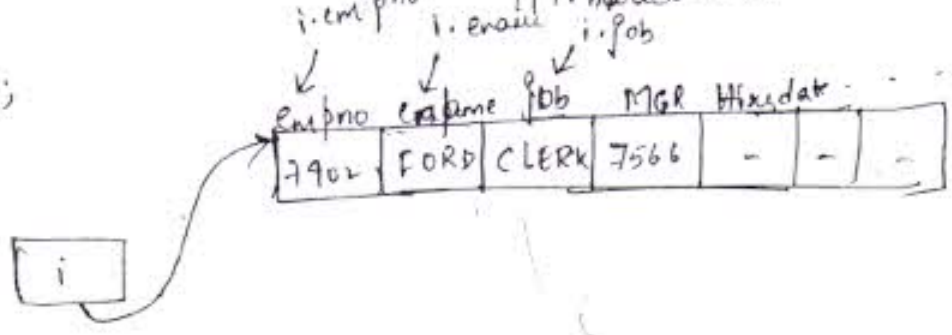
It is also called as record type variable

This variable is represented using %rowtype

It is also same as structures in c language.

Syntax :- variablename tablename %rowtype;

```
declare
i emp %rowtype;
begin
select ename, sal, hiredate into i.ename, i.sal,
i.hiredate from emp
where empno = &no;
dbms_output.put_line (i.ename || ' ' || i.sal || ' '
                      || i.hiredate);
end;
/
```



```
declare
 i emp%rowtype
begin
select * into i from emp
 where emp=&no;
 dbms_output.put_line (i.ename||' '||i.sal||' '||i.deptno);
 end;
 /
```

Conditional Statements

1.) if
2.) if ... else
3.) elsif — elsif

syntax of if :

```
if Condition then
 statements;
 end if;
```

2.) if .... else

syntax :

```
if Condition then
 statements;
else
 statements;
 end if;
```

3.) elsif : to check more no. of conditions we use elsif

syntax :-

```
if Condition1 then
 statements;
elsif Condition then
 statements;
```

```
elsif Condition 3 then
  statements;
  . . . . .
else
statements;
end if;
```

27/8/2012

```
declare
v_deptno number(10);
begin
select deptno into v_deptno
from dept where deptno: &no;
  if v_deptno = 10 then
dbms_output.put_line ('ten');
 elsif v_deptno: 20 then
dbms_output.put_line ('twenty');
 elsif v_deptno = 30 then
dbms_output.put_line ('thirty');
 else
dbms_output.put_line ('others');
 end if;
 end;
 /
```

Enter value for no: 10 ten

Enter value for no: 40 others

Enter value for no: 80

error: no data found.

Note1:
If a pl/sql block contains select into... clause & also requested data not available in a table oracle server returns an error as

ora-1403 : no data found

Note2: If a pl/sql block contains dml statements & also if requested data not available in a table oracle server doesnot returen any error.
To handle these type of blocks we use implicit cursor attributes

Ex:
```
begin
delete from emp
where ename = 'wwww';
end;
/
```
pl/sql procedure succesfully completed.

Note3: whenever select.... into clause try to return more than one value/record oracle server returns an error as

ora-1422 : Exact fetch returns more than one requested no of rows

Control ~~Statements~~ (loops)

→ There are 3 types of control statements supported by pl/sql

1.) simple loop
2.) while loop
3.) for loop

① Simple Loop :

This is also called as infinite loop
Here body of the statements are executed repeatedly

Syntax : loop
statements;
End loop;

Ex: begin
loop
dbms_output.put_line ('welcome');
end loop;
end;
/

To exit from infinite loop we use following two methods.

* method 1:
syntax : exit when true condition;

Ex:
declare
n number(10) := 1;
begin
loop
dbms_output.put_line (n);
exit when n>=10;
n := n+1;
end loop;
end;
/

n: 1 2 3 4 5 6 7 8 9 10
1>=10 false
n: 1+1=2

2>=10
n: 2+1=3

method: using if.

syntax: if <condition> then
exit;
end if;

Ex: declare
n number(10):=1;
begin
loop
dbms_output.put_line(n);
if n>=10 then
exit;
end if;
end n:=n+1;
end loop;
end;
/

2.) While loop :-
Here body of the stmts are executed repeatedly until condition is false.

syntax :- while (condition)
loop
stmts;
end loop;

Ex: declare
n number(10):=1;
while (n<=10)
loop
dbms_output.put_line(n);

```
n:=n+1;
end loop;
end;
/
```

3) for loop :-

Syntax :-

```
for index/loop variable name in lowerbound..upperbound
loop
statements;
end loop;
```

→

```
declare
n number(10);
begin
for n in 1..10
loop
dbms_output.put_line(n);
end loop;
end;
/
```

(without declaring a variable)

→

```
begin
for n in 1..10
loop
dbms_output.put_line(n);
end loop;
end;
/
```

```
begin
for n in reverse 1..10
loop
dbms_output.put_line(n);
end loop;
end;
/
```

non pl/sql variable (or) bind variable :

→ These variables are session variables
→ These are created at host environment
we can also use these variables in pl/sql to execute procedures having out parameters
These variables are also called as host variables

step 1 : Creating a bind variable

syntax : sql> variable variablename datatype ;

step 2 : using bind variable

syntax : :variablename

step 3 : display value from bind variable

syntax : print variablename ;

```
sql > variable g number ;
```

~~a number(10)~~

```
sql > declare
        a number(10) := 500;
      begin
        :g := a/2;
      end;
      /
sql > print g;
```

Result
```
  G
---
250
```

28/8 Flow

Cursor

→ It is a private sql memory area, which is used to process multiple records and also this is record by record process

→ There are two types of static cursors supported by oracle

1) Implicit cursor
2) Explicit Cursor

Implicit Cursor:

for sql stmts, returns single record is called Implicit Cursor.

Implicit Cursor memory area is also called as sql area

(or) Context area

Ex: declare
v_ename varchar2(20);
v_sal number(10);
begin
select ename, sal into
v_ename, v_sal from emp where empno = &no
dbms_output.put_line(v_ename || ' ' || v_sal);
end;
/

Enter value for no : 7902
FORD 5100

This stmt is used in executable section of the pl/sql block.

*Note

Whenever we are opening the cursor implicitly, cursor pointer points to the first record of the cursor.

3.) Fetch: (fetching from Cursor)

This stmt is used to fetch data from cursor memory area into pl/sql variables

Syntax: fetch cursorname into variable1, variable2 ...;

4.) Close:

Whenever we are closing the cursor all the resources allocated from cursor memory area are released

Syntax: close cursorname;

```
declare
Cursor c1 is select ename, sal from emp;
  v_ename  varchar2(10);
  v_sal    number(10);
begin
open c1;
fetch c1 into v_ename, v_sal;
dbms_output.put_line(v_ename||' '||v_sal);
 fetch c1 into v_ename, v_sal;
 dbms_output.put_line(v_ename||' '||v_sal);
 close c1;
 end;
/
```

Implicit Cursors are simple pl/sql programs which contains either select.... into clause or dml stmts but there all dml stmts are processed at a time

Explicit Cursors: for sql stmts, returns multiple records is called explicit Cursors

Explicit Cursor memory area is also called as active set area

Explicit Cursor life cycle:

1.) Declare
2.) Open
3.) Fetch
4.) close

Declare:

In declare section of the pl/sql block, we are defining Cursor memory area using following syntax:

syntax: Cursor Cursorname is select * from tablename where Condition;

Ex: declare
Cursor C1 is select * from emp
where sal > 3000;

Open: whenever we are opening the Cursor, then pl/sql runtime engine fetch the data from table into Cursor memory area because whenever we are opening the Cursor, then only select stmts are executed

syntax: Open Cursorname;

29/8/2012
Every Explicit Cursor having following four attributes.

(i) %. not found
(ii) %. found
(iii) %. is open
(iv) %. Row Count

when we are using these attributes in plsql block we must specify cursorname using following syntax

Syntax: Cursorname %. attribute name

Except %. rowCount all other Cursor attributes returns boolean value, either true (or) false whereas %. row Count attribute returns number data type.

%. not found: This Cursor attribute returns boolean value either true (or) false. i.e this attribute returns true if Cursor does not contain data after fetching records from the Cursor

write a pl/sql Cursor program, display all empnames, salaries from emp using %. not found attribute.

```
declare
Cursor c1 is select * ename, sal from emp;
v_ename varchar2(10);
v_sal   number(10);
begin
open c1;
loop
```

```
fetch c1 into v_ename, v_sal;
 exit when c1%notfound;
 dbms_output.put_line (v_ename||' '||v_sal);
 end loop;
  close c1;
  end;
  /
```

write a pl/sql Cursa program display first five highest sal employees using %rowcount attribute.

```
declare
Cursa c1 is select sal from emp;
  order by sal desc;
  v_sal number(10);
  begin
  open c1;
  loop
  fetch c1 into v_sal;
  exit when c1%rowcount >= 5
  dbms_output.put_line (v_sal);
    end loop;
    close c1;
    end;
    /
```

write a pl/sql Cursa program display even no. of records from emp table using %rowcount attribute.

```
declare
Cursa c1 is select ename, sal from emp;
```

```
v_ename   varchar2(10);
v_sal     number(10);
Begin
 Open c1;
 loop
fetch c1 into v_ename, v_sal;
exit when c1%notfound
   if mod(c1%rowcount, 2)=0 then
dbms_output.put_line(v_ename||' '||v_sal);
 end if;
 end loop;
  close c1;
  end;
```

%rowcount: Always %rowcount attribute returns "number" data type i.e this attribute stores number of records number fetched from the cursor

Ex: declare
cursor c1 is select ename, sal from emp;
v_ename varchar2(10);
v_sal number(10);
begin
open c1;
fetch c1 into v_ename, v_sal;
dbms_output.put_line(v_ename||' '||v_sal);
dbms_output.put_line('number of records fetched from cursor'||' '||c1%rowcount);
close c1;
end;
/

o/p: SMITH 8500
ALLEN 8500

whenever we are creating a cursor automatically 4 variables are allocated along with active set area. These variables are identified with cursor name along with attributename. These variables stores either number(01) boolean value

20/8/2012

```
SQL> Create table target (name varchar2(10), sal number(10));

SQL> declare
     Cursor c1 is select * from
     emp where sal > 3000;
      i emp%rowtype;
      begin
      open c1;
      loop
      fetch c1 into i;
      exit when c1%notfound;
       Insert into target
        value (i.ename, i.sal);
        end loop;
        close c1;
        end;
        /
SQL> select * from target;
```

Eliminating explicit Cursor life cycle:

→ using Cursor for loops we can eliminate explicit Cursor life cycle i.e within pl/sql blocks we are not allowed to use open, fetch, close statements

→ when we are using Cursor for loops internally pl/sql runtime engine takes open, fetch, close statements.

Cursor for loop :- ~~for~~ ~~Cursorname in~~

syntax

```
for loop index variablename in Cursorname
loop
statements;
end loop;
```

this loop is used in executable section of the pl/sql block.

Note:

→ In cursor for loop index variable implicitly behaves like a record type variable (% row type)

Ex:

```
declare
Cursor c1 is select * from emp
where sal > 2000;
begin
for i in c1
loop
dbms_output.put_line (i.ename || ' ' || i.sal);
end loop;
end;
```

[* Here the index variable need not be declared explicitly as it is declared implicitly]

→ ~~without~~

Note:

→ We can also eliminate declare section of the cursor using cursor for loop. in this we use select statement in place of cursor name in cursor for loop

Syntax:

```
for variable name in (select statement)
loop
stmts;
end loop;
```

Ex: begin
for i in (select * from emp where sal > 2000).
loop
dbms_output.put_line (i.ename || ' ' || i.sal);
end loop;
end;

Parameterized Cursors:

→ We can also pass parameters to cursors as same as sub program in parameters

→ In these type of cursors we must define formal parameters in cursor definition and actual parameters in open stmt

(1) Syntax:- Cursor cursorname (parameter name datatype)
formal parameter declaratn
is select * from tablename
where columnname parameter name;

(1) Syntax:- open cursorname (actual parameter);

Note:- In oracle whenever we are defining parameters to cursors, procedures & functions we are not allowed to use datatype size in formal parameter declaration.

→ formal parameter.

declare
Cursor c1 (p_deptno number) is
select * from emp where
deptno = p_deptno;
i emp %rowtype;

```
begin
open c1(10);        actual parameter
loop
fetch c1 into i;
exit when c1%notfound;
dbms_output.put_line(i.ename||' '||i.deptno);
end loop;
close c1;
end
```

Q) write a pl/sql program using parameterized cursors display following static report from emp table

employees working as managers

JONES
CLARK
BLAKE

employees working as analysts

SCOTT
FORD

```
declare
Cursor c1 (p_job varchar2) is
select * from emp where
    job: p_job;
%emp% rowtype;
begin
open c1('Manager');
dbms_output.put_line (employees working as managers);
loop
fetch c1 into i;
```

```
exit when c1%
exit when c1%notfound;
dbms_output.put_line(i.ename);

end loop;
close c1;
open c1('Analyst');
dbms_output.put_line('Employees working as analyst');
loop
fetch c1 into i;
exit when c1%notfound;
dbms_output.put_line(i.ename);
end loop;
close c1;
End;
/
```

31/8/2012

```
declare
Cursor c1 is select * from emp where sal>1000
i emp%rowtype
  begin
  open c1;
  loop
  fetch c1 into i;
  exit when c1%notfound;
  dbms_output.put_line(i.ename||' '||i.sal);
  end loop;
  close c1; end;
```

Note: Before reopening the cursor we must close the cursor otherwise oracle server returns an error ora-6511: cursor already open

Note: If you do not open the cursor in pl/sql block oracle server returns an error

ora-1001 invalid cursor

Converting parameterised cursor program to cursor for loop -

Ex: declare
cursor c1 (p_deptno number)
is select * from emp where
deptno = p_deptno;
begin
for i in c1 (10 (or) &no)
loop
dbms_output.put_line (i.ename||' '|| i.deptno);
end loop;
end;

Note: In parameterised cursor we can also pass default value using default (or) := operator

Syntax: parametername datatype default (or) [:=]
actual value;

Ex: declare
cursor c1 (p_deptno number default 20) is
select * from emp where deptno = p_deptno;
begin

```
for i in c1(10)
loop
dbms_output.put_line (i.ename || ' ' ||i.deptno);
end loop;

end
/
```

=) write a pl/sql program display employee details from emp table based on deptnos from dept table.

```
declare
Cursor c1 is select deptno from dept;
Cursor c2 ( p_deptno number) is select * from
    emp where deptno: p_deptno;
begin
for i in c1
loop
dbms_output.put_line (deptno is || ' ' || i.deptno);
for j in c2 (i.deptno)
loop
dbms_output.put_line (j.ename || ' ' || job || ' ' ||
                                    j.deptno);

end loop;
end loop;
end
```

write a pl/sql prog. to modify salaries of the employees from emp table using following conditions.

1) If job : 'CLERK' then increment salary to 100
2) If JOB : 'SALESMAN' then increment sal to 200
3) If job : 'ANALYST' then increment sal → 100

Ex: declare
Cursor c1 is select * from emp;
i emp%rowtype;
begin
open c1;
loop
fetch c1 into i;
exit when c1%notfound;
if i.job = 'CLERK' then
update emp set sal = i.sal+100
where empno = i.empno;
elsif i.job = 'SALESMAN' then
update emp set sal = i.sal-200
where empno = i.empno;
elsif i.job = 'ANALYST' then
update emp set sal = i.sal+100;
where empno = i.empno;
end if;
end loop;
end;
/

1/3/2012 Where Current of, for update clauses used in cursors.

(or)

update, delete statements used in Cursors.

Generally whenever we are performing updation, deletion operations automatically all databases uses locking mechanism & if we want to perform locks before update (or) before delete all databases introduce explicit Cursor locking mechanism & in this case we must use "for update clause" in cursor select statement.

Syntax: Cursor Cursorname is select * from tablename where Condition for update;

when we are specifying for update clause & also whenever we are opening the cursor then only locks are established i.e all databases systems internally uses exclusive locks when we are opening the cursor.

Where Current of

Where Current of clause used in update, delete statements only Where Current of clause internally uses rowid's in a table & thats why it is uniquely identify a record in each clause

syn:- 1) Update tablename set Columnname = new value where Current of Cursorname;

2) Delete from tablename where Current of Cursorname;

Note: whenever we are using a where current of clause we must use "for update clause" after processing the records using locking mechanism we must release the locks using Commit

write a query job clerk sal is incremented by 100.

Ex:
```
declare
Cursor c1 is select * from emp for update;
i emp%rowtype;
begin
open c1
loop
fetch c1 into i;
exit when c1%notfound;
if i.job = 'CLERK' then
update emp set sal = i.sal+100
    where current of c1;
end if;
end loop;
Commit;
close c1;
End;
```

Write a pl/sql programme using cursor locking mechanism raise 5% salary of the employees who are working under king (king as manager) from emp table.

Ex:
```
declare
cursor c1 (p_mgr number) is select sal from
```

```
emp where mgr: p-mgr for update;
        v-mgr number(10):
begin
select empno into v-mgr from emp where ename='KING';
for i in c1 (v-mgr)
loop
update emp set sal = sal * 0.15 where current of c1
end loop;
commit;
end;
```

Implicit Cursor attributes:

When a pl/sql block contains select into clause (or) when a pl/sql block contain pure dml stmts oracle server automatically creates a memory area, this memory area is also called as sql area (or) enter area (or) implicit cursor, along with sql area four variables are automatically created these variables are identified through attribute name along with sql keyword

Thats why implicit cursor internally has 4 attributes

(i) sql % found
(ii) sql % not found
(iii) sql % is open
(iv) sql % Row Count.

These Cursor attributes return values when we are using procedural stmts in pl/sql blocks.

***Note: Always sql%isopen returns false (boolean value) where as sql%rowcount returns "number datatype".

Ex:
```
begin
delete from emp where ename='sunday';
                                (*pure dml stmt*)
if sql%found then
dbms_output.put_line('record exist and is deleted');

end if;
 if sql%found then
 dbms_output.put_line('record doesnot exist');

 end if;
  end;
/
```
o/p: record does not exist.

Ex: of rowcount:
```
begin
update emp set sal=sal+10 where job='CLERK';
dbms_output.put_line('affected number of clients are'
                     ||' '|| sql%rowcount);

end;
/
```
o/p: affected no. of Clerks are: 4.

3/7/2017

Exceptions

→ Exception is an error occured during runtime
→ whenever runtime error occured we an appropriate ~~name~~ in exception name in exception handling.
→ There are 3 types of exceptions supported by pl/sql
  → predefined exceptions
  → user defined exceptions
  → unnamed exceptions

Predefined Exception

→ Oracle defined 20 predefined exception names for regularly occured runtime errors
→ whenever error occured we the predefined exception name in an exception handler under exception section

Syntax:
```
when predefined exception name1 then
statements;
when predefined exception name2 then
statements;
- - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - -
when others then
statements;
```

Some important predefined exceptions
1) no_data_found
2) Too_many_rows
3) Zero_divide
4) Invalid_cursor
5) Cursor_already_open
6) Invalid_number
7) Value_error

1) no_data_found

→ when a pl/sql block contains select into clause & also if requested data not available in a table oracle server returns an error as ora-1403 : no data found

to handle this error we are using no_data_found exception name

```
declare
  v_ename varchar2(10);
  v_sal   number(10);
begin
select ename sal into
v_ename, v_sal from emp where
 empno : &no ;
 dbms_output.put_line ( v_ename || ' ' v_sal);

exception
  when no_data_found then
  dbms_output.put_line ('your employee does not exist');

  end;
  /
```

o/p: Enter value for no : 7902

FORD 5500

Enter value for no : 1111

your employee does not exist

2.) Too_many_rows :-

→ whenever select into clause try to return more than one value oracle server returns an error

ora-1422 : exact fetch returns more than requested no. of rows

→ To handle this error we using

Too_many_rows

Ex: declare
v_sal number(10);
begin
select sal into v_sal from emp;
dbms_output.put_line (v_sal);
exception
when too_many_rows then
dbms_output.put_line ('not to return more rows');
end;
/

o/p: not to return more rows

3.) Zero_divide :

ora-01476 : divisor is equal to zero.

4.) Invalid_Cursor ;

Ex: a number(10);
b number(10);
c number(10);
begin
a := 5;
b := 0;

```
c := a/b;
dbms_output.put_line(c);
exception
 when zero_divide then
dbms_output.put_line('b cannot be 0');

end;
/
```

4) Invalid_Cursor:

→ when we are not opening the cursor in pl/sql block oracle server return an error as ora-1001 : invalid cursor

To handle this error we can use Invalid_cursor exception name

Ex:
```
declare
Cursor c1 is select * from
emp where sal>2000;
 i emp%rowtype;
 begin
 loop
 fetch c1 into i;
 exit when c1%notfound;     i.ename||' '||i.sal);
 dbms_output.put_line('~~first we must open~~ The cursor');
 end loop;
 close c1;
 exception
  when invalid_cursor then
dbms_output.put_line('first we must open the
                      cursor');

  end;
  /
```

5.) Cursor_already_open

Whenever we are reopening the cursor without closing the oracle server returns an error as

ora-6511 : Cursor already open

To handle this error we are using cursor_already_open

Ex:

```
declare
  Cursor c1 is select * from emp
  where sal>2000;
    i emp%rowtype;
    begin
    open c1;
    loop
    fetch c1 into i;
    exit when c1%notfound;
    dbms_output.put_line(i.ename||' '||i.sal);
    end loop;
    open c1;
    loop    exception
            when cursor_already_open then
    fetch c1 into i;
    exit;   dbms_output.put_line('we must close the
                cursor before reopening');
    end;
    /
```

Invalid_number, value_error

→ whenever we try to convert string type to number type oracle server returns two types of errors they are

Invalid number, value error

→ when a pl/sql block contains sql stmts and also if we try to convert string type to number type oracle server return - s an error

ora-1722 invalid number

→ To handle this error we use invalid_number exception name.

4/9/2017.

```
begin
insert into
emp (empno, sal) values (1, 'abc');
exception
when invalid_number then
dbms_output.put_line ('insert appropriate data');
end;
/
```

o/p: insert appropriate data.

```
declare
  v_deptno  varchar2(10) := '&deptno';
  v_dname   varchar2(10) := '&dname';
  v_loc     varchar2(10) := '&loc';
```

```
begin
insert into dept
values (v_deptno, v_dname, v_loc);

exception
 when invalid_number then
dbms_output.put_line ('enter proper data');

end;
/
```

** Value_error

→ when a pl/sql block contains plsql stmts & also when we are trying to convert string type to number type oracle server raises an error

ora-6502 : numeric or value error : character to number conversion error.

To handle this error we use value_error exception name.

Ex:

```
declare
z number(10);
begin
z:='4x'+'84';
dbms_output.put_line (z);
exception
when value_error then
dbms_output.put_line ('enter proper data');
 end;
/
```

Note:

value error also occurs when we try to store more data than the specified data type size in a variable. In this case oracle server returns an error ~~ORA6502~~.

ora - 6502 : numeric or value error character string buffer too small

To handle this error also we use value_error exception.

Ex

```
declare
z varchar2(3);→ ~~dec~~ 'given 3 bytes
                but abcd is more than 3 bytes
begin           hence generates an error
z:= 'abcd';
exception
when value_error then
dbms_output.put_line('invalid string length');
end;
/
```

```
declare
z varchar2(3) := 'abcd';   ← This can be handled by using exception in outer block.
begin
dbms_output.put_line(z);
exception
when value_error then   } → even though the exception is handled if do not provide invalid string length instead it produces an error ie ora 6502 character string buffer too small. Hence error is produced in declare section
dbms_output.put_line('invalid string length');
end;
exception
when value_error then
```

dbms_output.put_line ('invalid string length handled using outer blocks');

end;

/

O/P invalid string length handled using outer blocks.

Exceptions also raised either in executable, declare or in exception section is called exception propagation.

** → when exceptions are raised in executable section those exception are handled using inner block of outer block. Whereas when exceptions are raised in declare or in exception section, those exceptions must be handled using outer blocks only

User defined Exceptions

→ We can also create our own exceptions and also raise whenever necessary in pl/sql blocks

1.) declare
2.) Raise
3.) handling exceptions.

declare :- If in declare section of pl/sql block we are creating our own exception name using exception predefined type

Syntax :- user defined exception name exception;

Ex: declare
a exception;

Raise :- using raise stmt explicitly we are raising user defined exception either in executable section or in exception section.

syntax: raise user defined exceptionname;

If we are specifying raise stmt also internally pl/sql runtime engine will raise that exception.

3) handling exception: we can also handle user defined exceptions same as predefined exceptions using exception handler under exception section

6/9/2012.

* write a pl/sql program raise a user defined exception based on the system date

```
SQL> declare
       e exception;
     begin
       if to_char(sysdate,'DY') = 'WED' then
         raise e;
       end if;
     exception
       when e then
       dbms_output.put_line('my exception raised today');
     end;
     /

SQL> declare
       v_sal number(10);
       e exception;
     begin
       select sal into v_sal from emp
```

```
where empno = 7902 ;
if v_sal > 2000 then
raise e;
exception
when e then
dbms_output.put_line ('salary already high');

end;
/
```

Note: If we want to transfer control to the no of blocks without checking each individual stmt then only use user defined exception.

→ If we want to stop exception propagation the we use user defined exceptions.

Exception raised in executable section

i) using inner blocks:

```
sql> declare
     e exception;
     begin
     raise e;
     exception
     dbms_output.put_line ('handled using inner blocks');

     end;
     /
```

1.) using outer blocks

```
sql> declare
       z exception;
     begin
     begin
      raise z;
     end;
    exception
     when z then
    dbms_output.put_line('exception handled using outer
                          blocks');
    end;
    /
```

2) Exception raised in exception section

must be handled by using outer blocks only

```
declare
 z1 exception;
 z2 exception;
begin
 begin
 raise z1;
exception
 when z1 then
 dbms_output.put_line('z1 handled');
   raise z1;
```

```
when z1 then
dbms_output.put_line ('z1 handled');
end;
exception
when z2 then
dbms_output.put_line ('z2 handled using outer block');
end;
/
```

6/9/2012

Raise Predefined Exception :

We can also raise predefined exceptions using raise statement.

Syntax: raise predefined exception name ;

```
declare
cursor c1 is select * from emp
  where job='10';
r emp%rowtype;
begin
open c1;
fetch c1 into r;
if c1%rowcount=0 then
raise no_data_found;
end if;
close c1;
```

```
dbms_output.put_line (v_sal);
exception
when others then
dbms_output.put_line (sqlcode);
end;
/
```

-1422

Ex:

```
declare
e exception;
begin
raise e
exception
when e then
dbms_output.put_line (sqlcode);
dbms_output.put_line (sqlerrm);
end;
/
```

o/p: 1
user defined exception

Note:

We can also use these functions in insert stmt indirectly
In this case we need declare variables and assigns values into variables and use these variables in insert stmt

exception
when no_data_found then
dbms_output.put_line('your job not found in my table');
end;

Error trapping functions:

1) sqlcode
~~2) sqlerrm~~
2.) sqlerrm.

Oracle provided 2 error trapping functions they are sqlcode, sqlerrm. these functions are used in exception section.

Sqlcode returns number whereas sqlerrm returns error number with error message

Sqlcode return value	Meaning.
0	No errors
-ve	oracle errors
100	no data found
1	user defined exception

declare
v_sal number(10);
begin
select sal into v_sal from
emp;

write a pl/sql program to store error number, error msg. in a table of the particular exception.

```
Sql> Create table b1 (erno number(10), erm varchar2(100));

Sql > declare
      v_sal number(10);
      v_erno number(10);
      v_msg varchar2(100);
      begin
      select sal into v_sal from emp;
      exception
        when others then
        v_erno := sqlcode;
        v_msg := sqlerrm;
        insert into b1 values (v_erno, v_msg);
        end;
        /
    Sql> select * from b1;
```

→ raise_application_error ()

This is a predefined functn which is used to display user defined exception messages in more descriptive form i.e if we want to display user defined exception messages as same as oracle error displayed format then only we use this function

→ This function is used in either executable or in exception section.

This function accepts two parameters

Syntax: Raise_application_error (errornumber, message)

errornumber must be b/w (-20000 to -20999)

message (upto 512 characters)

```
declare
 v_sal number(10);
 e exception;
 begin
 select sal into v_sal from
   emp where empno = 7902;
   if v_sal > 2000 then
   raise e;
   else
    update emp set sal: sal+1000
      where empno = 7902;
    end if;
    exception
     when e then
     raise_application_error (-20432 ; 'salary already big');
    end;
    /
```

O/p : ORA-20432 : salary already high

Ex:

Note: generally raise_application_error is used in triggers because whenever condition is true it returns a msg and also it prevents invalid data entry according to conditions.

10/9/2012

Unnamed Exceptions :-

→ If we want to handle other than oracle 20 predefined exception name error numbers we use unnamed method

In this method we create our own exception name & associate this exception name with appropriate error number using ex

exception_init ()

because oracle provided exception names for regularly occured 20 runtime errors.

Syntax : pragma Exception_init (userdefined Exception name, errorno);

Here pragma is a compiler directive i.e at the time of compilation only pl/sql runtime engine associate error number with exception name

This function is used in declare section of the pl/sql block.

Ex: declare
x exception;
pragma Exception_init (x, -1400);

```
begin
insert into emp (empno, ename) value (null, 'murali');
exception
  when z then
dbms_output.put_line ("Cannot insert null");

end;
/
```

```
declare
z exception;
pragma exception_init (z, -2292);
 begin
delete from dept where deptno=10;

  exception
  when z then
dbms_output.put_line ("we cannot delete records from master
                       record table");

  end;
  /.
```

write a pl/sql program handle -2291 error using.
exception_init () from emp. dept table

```
  declare
  z exception
  pragma exception_init (z, -2291);
    begin
```

```
insert into emp (empno, deptno) values (1, 70);

exception
when z then
dbms_output.put_line ('Cannot insert other than primary
                       key values into foreign key');
end;
/
```

if we try to insert other than pkey values into foreign key we get this error

Sub-Programs

These are named pl/sql blocks which is used to solve some particular task

There are 2 types of sub programs supported by oracle

(may or may not return a value)

1) Procedures (returns multiple values)
2) Functions
   (must return a value)

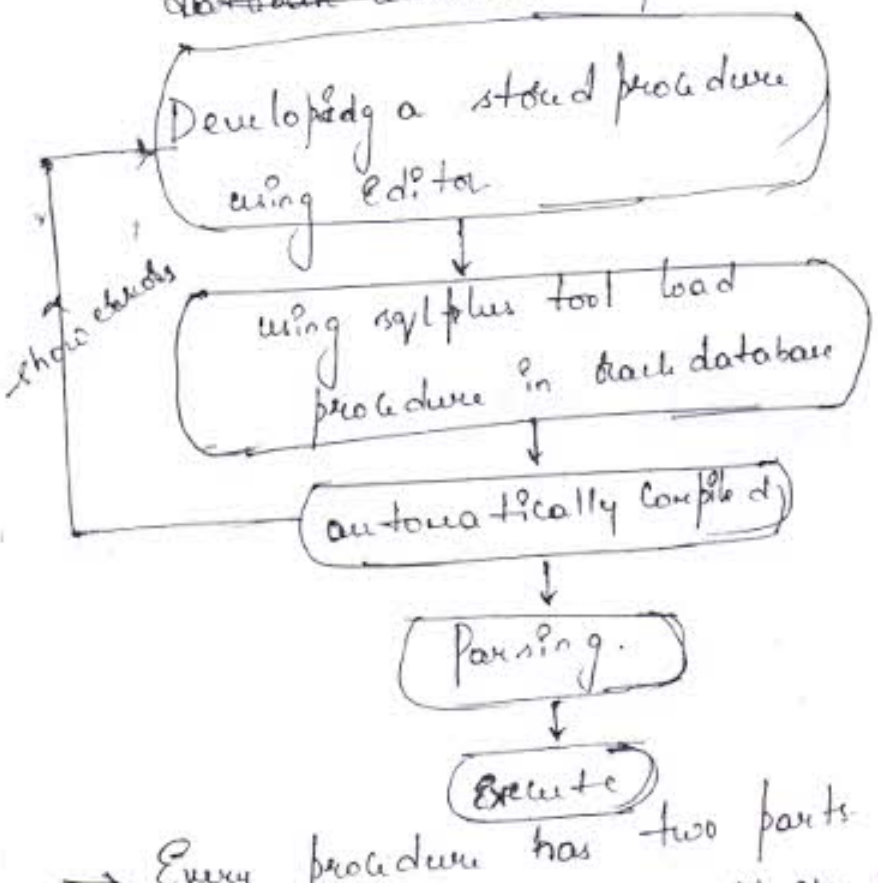
Procedures

It is a named pl/sql block which is used to solve some particular task & also procedure may or may not return a value.

→ when we are using create or replace keyword in front of the procedure, those procedures automatically permanently stored in database

thats why they procedures are also called as stored procedures

→ Generally procedures are used to improve performance of the applicatn because procedures internally has one time compilation i.e whenever we are submitting procedure into database automatically those procedures are compiled.



→ Every procedure has two parts

1) Procedure specification
2) Procedure body

In procedure specificatn we specify name of the procedure & type of the parameters

whereas in procedure body we write solve actual task.

Syntax :- Procedure specification { Create or replace procedure procedurename (formal parameters)

Procedure body:

- Is | As
- variable declarations, Cursors, User defined exceptions;
- Begin
- ....
- (Exception)
- ....
- end (Procedure name)

[Parameter name [mode]
- in
- out
- in out .

To view errors:

```
Sql > show errors;
```

Executing a Procedure:

Method 1:

```
Sql > exec procedure-name (actual parameters);
```

Method 2:

using anonymous clause

```
Sql > begin
      procedure name (actual parameters);
      end;
      /
```

Method 3

```
Sql > Call procedurename
      procedurname (actual parameters);
```

11/8/2012

Write a pl/sql stored procedure for passing employee number as a parameter., display name of employee & his salary from emp table

```
Create or Replace procedure p1 (P_empno number)
is
v_ename varchar2 (10);
v_sal   number (10);
begin
select ename , sal into v_ename , v_sal from
 emp where empno = P_empno;
dbms_output.put_line (v_ename || ' ' || v_sal);
  end;
  /
```

Execution

method 1 :-

```
sql> exec p1 (7902);
```

FORD 3000

Method 2 : (using anonymous blocks)

```
SQL> begin
     p1 (7566);
     end;
```

JONES 1975

Method 3 :

```
SQL> Call p1 (7902);
```

FORD 3000

Procedure body:

Is | As
Variable declaration, Cursors, User defined exceptions;
Begin
.....
(Exception)
.....
end (Procedure name)

(Parameter name [mode]
  - in
  - out
  - in out .

To view errors:

```
sql > show errors;
```

Executing a Procedure:

Method 1:

```
sql > exec procedure-name (actual parameters);
```

Method 2:

Using anonymous clause

```
sql > begin
      procedure name (actual parameters);
      end;
      /
```

Method 3

```
sql > Call procedurename
      procedurename (actual parameters);
```

11/9/2012

Write a pl/sql stored procedure for passing employee number as a parameter, display name of employee & his salary from emp table

```
Create & Replace procedure p1 (P_empno number)
is
v_ename varchar2 (10);
v_sal   number (10);
begin
select ename, sal into v_ename, v_sal from
 emp where empno = P_empno;
dbms_output.put_line (v_ename || ' ' || v_sal);
  end;
  /
```

Execution

method 1 :-

```
sql> exec p1 (7902);
```

FORD 3000

Method 2 : (using anonymous blocks)

```
SQL> begin
     p1 (7566);
     end;
```

JONES 1975

Method 3 :

```
SQL> Call p1 (7902);
```

FORD 3000

→ write a pl/sql stored procedure for passing deptno as a parameter, display employee details of particular dept from emp table.

ALLEN 2000 10
WARD 3000 10
KING 5000 10

```
Create or Replace Procedure P1
is
  V_ename  varchar2(10);
  V_sal    number(10);
begin
select * from emp where deptno = &deptno;

Create or Replace Procedure P1 (P_deptno number)
is
Cursor C1 is
select * from emp where deptno = P_deptno;
I Emp%rowtype;
Begin
open C1;
Loop
fetch C1 into I;
exit when C1%notfound;
dbms_output.put_line(i.ename||' '||i.sal||' '||i.deptno)
End Loop;
Close C1;
end;
/
```

SQL> exec p1(10);

Parameters used in Procedures:

→ Parameters are used to pass values into procedure & return values from procedure

There are 2 types of parameters supported by procedures

1.) formal parameters
2.) Actual parameters.

Formal parameters: These are defined in procedure specification Sp specifies parameter name, mode of the parameter, data type

There are 3 types of Modes supported by formal parameters

1.) In Mode
2.) Out Mode
3.) In Out mode

1.) In Mode

: Syntax: parameter name [mode] datatype

By default mode is in mode, this mode is used to pass values into procedure body

This mode behaves like a constant in procedure body, using in mode we can also pass default values using default & := operator

write a pl/sql stored procedure, insert a record into dept-table using in parameters

```
Create or replace procedure p1 (p_deptno in number, p_name
        in varchar2, p_loc in varchar2)

is
begin
insert into dept
  values (p_deptno, p_dname, p_loc);
dbms_output.put_line ('record inserted through
                       procedure');

end;
/
```

```
Sql> exec p1(5, 'x', 'y');
```

There are 3 types of execution methods supported by in parameter

1.) positional notations Ex:
2.) Named notations
3.) Mixed notation

1.) positional notation

```
Ex: execp1(1, 'a', 'b');
```

2.) Named notation

```
Sql> execp1(p_dname => 'x', p_loc => 'y', p_deptno => 2);
```

3.) Mixed notation :
It is the combination of positional, named notation

Rule : After positional matching there can be all named notations. but after named notation there cannot be positional

Ex:

Create or replace procedure p1 (p_deptno number, p_dname varchar2, p_loc varchar2 default 'amerpet').

is
begin
insert into dept
values (p_deptno, p_dname, p_loc);

end;
/

Out

This mode is used to return values from procedure body
This mode internally behaves like a uninitialised variable
in procedure body. Here explicitly we must specify out
keyword

Create or replace procedure p1 (a in number, b out number)

is
begin
b := a * a;
end;
/

a*b*b — not possible because a is a const we cannot store a value into const.

→ When a subprogram contains out, inout parameter, those sub programs are executed using following two methods.

Method 1.) using bind variable

Method 2: using anonymous blocks.

Method 1: using bind variable

```
Sql> variable z number;
sql> exec p1 (7, :z);
Sql> print z;
```

~~7/64~~

```
O/P:  z
      64
```

12/9/2012

Method 2: using anonymous block.

```
declare
x number(10);
begin
p1(7, x);
dbms_output.put_line(x);
end;
```

write a pl/sql stored procedure for passing emp name as in parameter return salary of the employee using out parameter from emp table

```
Create(or) replace procedure p1 (p_name in varchar2,
                                 p_sal out number);
is
begin
cursor
select sal into "p_sal" from emp where ename = p_ename;
end;
/
```

Execution

method 1 : using bind variable

```
variable z number;
exec p1('SMITH', :z);
```

pl/sql procedure successfully completed

```
Sql> Print z;
```

o/p

z
1200

method 2 : using anonymous block

```
declare
z number(10);
begin
p1('KING', z);
dbms_output.put_line(z);
end;
/
```

o/p : 5000.

→ write a pl/sql stored procedure for passing deptno as a parameter returns number of employees for that particular dept using out parameter from emp table

```
Create or replace procedure p1(p_deptno in number,
                               p_no out number)
is
begin
select Count(*) into p_no from emp where
    deptno = p_deptno;
end;
/
```

Execution
using bind variable:
sql> variable a number;
sql> exec p1(10, :a);
o/p 3.

Inout: This mode is used to pass the values into procedure and return values from the procedure and also this mode behaves like a constant, initialised variable in procedure body. Here also explicitly we must specify in out keyword

Ex: Create (or) replace procedure p1 (a in out number)
is
begin [# Here 'a' behaves like a variable & constant #]
a := a * a;
end;
/

Execution
method: using bind variable:
sql> variable z number;
sql> exec :z := 6
sql> exec p1(:z);
sql> print z;
o/p: z
36

Method: using anonymous block
sql> declare
z number(10) := &x;
begin
p1(x);
dbms_output.put_line(z);
end;

write a pl/sql stored procedure for passing empno as a parameter return salary of the employee using inout parameter from emp table

Create or replace procedure p1 (p_a in out number)
is
begin
select sal into p_a from emp where
empno = p_a;
end;
/

Execution
using bind variable
sql> variable z number;
sql> exec :z := 7902;
sql> exec p1(:z);
sql> print z;
o/p z
2500

13/9/2012

Pass by value, Pass by reference :

Whenever we are using modular programming all types of Languages supports two types of passing parameter mechanisms they are

1) Pass by value.
2) Pass by reference

These two parameter mechanisms specify when we are modifying formal parameters, actual parameter may affect or may not
In pass by value method actual value does not change in calling program, because internally copy of the value are passed into called program.
If we want to change actual values corresponding to the formal parameters then we use pass by reference method

Oracle server supports these two parameter mechanisms when we are using parameters in sub programs.
by default all in parameters use pass by reference method
whereas by default all out parameters use pass by value method
→ whenever we are returning large amount of data using out parameter again copy of the values are created because out parameter internally uses pass by value

→ To overcome this problem, oracle introduced nocopy hint in procedure parameters.

Syntax: parametername out nocopy datatype

Ex: Create or replace procedure p1 (p1_ename in varchar2,
p_sal out nocopy number)

is
begin
select sal into p_sal from emp
where ename = p_ename;
end;
/

Autonomous transactions

→ There are independant transactions used in procedures, triggers
→ If a procedure behaves independently from main transaction, that procedure must have autonomous_transaction pragma, commit

Syntax: pragma autonomous_transaction;

This pragma is used in declare section of the procedure or declare section of trigger

Syntax: Create or replace procedure procedurename
(formal parameters)
is / as
pragma autonomous_transaction;
begin
---
commit;

[exception];

...

end;

Note: Before Oracle 8.1.6, when a procedure contains transactional commands, those transactional commands not only affects procedure transactn but also affects above the procedure transactn above the main program. To overcome the problem oracle introduced autonomous transaction in program units.

Using autonomous transaction

Ex: sql> Create table test (name varchar2(10));

```
sql> Create or replace procedure p1
     is
     pragma autonomous_transaction;
     begin
     insert into test value ('hyd');
     insert into test value ('mumbai');
     Commit;
     end;
     /
sql> begin
     Insert into test value ('India');
     p1;
     Rollback;
     end;
     /
```

o/p: select * from test;

hyd
mumbai

without using autonomous transaction

```
Sql> Create or replace procedure P1
     is
     begin
     insert into test values ('hyd');
     insert into test values ('Mumbai');
     commit;
     end;
     /

Sql> begin
     insert into test values ('India');
     P1
     rollback;
     end;
     /
```

O/P: select * from test.

India
hyd
Mumba:

authid Current_user:

→ this clause is used in specification of the procedure, generally this clause is used to provide data security & pointer vice versa whenever we are reading data from table & perform some operations

If procedure has this clause that procedure gives privilege to another user also, another user cannot execute that procedure

Syntax: Create or replace procedure procedurename (formal parameters)

~~authid = use~~

```
authid current_user
is/as
- - - -
- - - -
begin
- - - -
[Exception]
- - - -
end;
/
```

giving privileges on procedure to user

grant execute on procedurename to user1, user2 ..... ;

14/9/2012 Ext

```
Sql > Create or replace procedure P1 (P_empno number)
authid current_user
is
v_ename varchar2(10);
v_sal   number(10);
begin
select ename, sal into v_ename, v_sal from emp
    where empno = P_empno;
dbms_output.put_line (v_ename || ' ' || v_sal);
end;
/
```

Sql> grant execute on P_1 to sunali;

sql> Conn sunali/sunali;

Sql> set serveroutput on;

sql> exec scott.P_1(7902);

Error : table or view does not exist.

Handled (&) unhandled Exceptions in Procedures :

Whenever we are calling inner procedures into outer procedures we must handle inner procedure exception, otherwise pl/sql runtime engine automatically calls outer procedure default exception handle.

Inner procedure :

```
Create or replace procedure P1 (a in number, b in number)
is
begin
dbms_output.put_line (a/b);
exception
when zero_divide then
dbms_output.put_line ('b cannot be zero');
end;
/
```

Outer Procedure

```
Create or replace procedure P2 &
is
begin
```

```
p1(5, 0);
exception
 when others then
dbms_output.put_line('any error');
 end;
 /
```

```
SQL> exec P1;
```

b cannot be zero

→ All stored procedure information stored under user_procedures, user_source data dictionary.

→ We can also drop procedures using

drop procedure procedurename.

Functions

Functions are used to solve some particular task and also functions return a value

functions also has two parts 1.) function specification
2.) function body.

In funct^n specification we specify name of the funct^n & type of the parameters

Syntax: Create or replace function
functionname (formal parameters)
Return datatype
Is/As
→ variable declarations, Cursor;

Begin

Return expression;

end [function name];

optional.

Executing a function.

Method 1:- select function name (actual parameter) from dual;

Method 2: using anonymous block.

Begin

Variable name := function name (actual parameters);

end;

Ex: Create or replace function f1 (a varchar2)

return varchar2 — [should n't be used as if is a specification]

is

begin

return a;

end;

Execution:

Method1: using select statement

select f1 ('hi') from dual;

o/p hi

Method2: using anonymous block.

declare

z varchar2(10);

begin

z := f1('hi');

dbms_output.put_line(z);

end;

write a pl/sql stored funct^n for passing number as a parameter return a message either even & odd based on that number.

```
Create or replace function f1 ( a . number, b number)
  return varchar2 (a in
  is
begin      (mod(a,2) : 0 )
  if (a+b = 0)
dbms_output . put_line ( even);
else
dbms_output . put_line (odd);
  return a to (even/odd) a.
  end;
```

```
Create or replace function f1 (a number)
  return varchar2
  is
begin
if (mod (a,2) = 0 then
  return 'even';
else
return 'odd';
  end if;
  end;
```

Execution

Method 1 :- select $f_1(4)$ from dual;

even

Method 2 :- using anonymous blocks

```
declare
x varchar2(10);
begin
x := f1(7);
dbms_output.put_line(x);
end;
```

o/p :- odd

Method 3 :- using bind variable

```
SQL> Variable z varchar2(10);
begin
:z := f1(9);
end;
/
SQL> print z
```

o/p :- odd

Method 4 :- Exec dbms_output.put_line(f1(4));

o/p even

Method 5 :-

```
begin
dbms_output.put_line(f1(5));
end;
/
```

odd

Note :- We can also use functions functionality within insert stmt.

```
Sql> Create table w1 (msg varchar2(10));
sql> Insert into w1 values (f1(6));
sql> select * from w1;
```

o/p Msg
Even

If we want max(sal) & max(sal)-sal, even by using group by it doesn't give correct result
hence to overcome this problem by using functions

Note: We can also use predefined aggregate functn within user defined functions and also we can use these user defined functions in same table or different table

Ex: Create or replace function f1

```
return number
is
v_sal number(10);
begin
select max(sal) into v_sal from emp;
return v_sal;
end;
```

Execution
```
Sql> select ename, sal, f1, f1-sal from emp;
/
```

15/9/2017.

write a pl/sql stored function for passing empno as a parameter return gross salary of the employee based on following conditions.

hra -> 10% of sal

da -> 20% of sal

Pf -> 10% of sal

gross: basic + hra + da - pf.

Create or replace function fn (p_empno number)

return number

is

v_sal number(10);

gross number(10);

hra number(10);

da number(10);

Pf number(10);

begin

select sal into v_sal from emp where empno = p_empno;

hra := v_sal * 0.1;

da := v_sal * 0.2;

Pf := v_sal * 0.1;

gross := v_sal + hra + da - Pf;

return gross;

End;

/

Execution

select fn(7902) from dual;

write a plsql stored functn for passing empno, date as parameters return number of years that employee worked based on the date from emp table.

```
Create or replace function f1
 return number
 is
 v_empno  number(10);
 v_date   date
 begin
 select * from emp where
  empno = v_empno, date = v_date;
 return v_empno;
 return v_date;
```

Create or replace function f1 (p_empno number, p_date date)

```
return number
 is
 x number(10);
 begin
 select months_between(p_date, hiredate)/12
      into x from emp where empno = p_empno;
 return round(x);
 end;
```

Execution

```
select empno, ename, hiredate,
  f1(empno, sysdate) || ' ' || 'years' "exp" from
  emp where empno = 7902;
```

Empno	Ename	Hiredate	exp
7902	scott	19-Apr-82	31 yrs

Note: Prior to Oracle 11g when we are calling sub program into select stmt, then we are not allowed to use named, mixed parameters i.e in that case we are only passing real parameters but in Oracle 11g we can use named, mixed notations when we are calling a subprogram in select stmt.

```
select empno, ename, hiredate,
  f1(p_empno => empno, p_date => sysdate) || ' ' || 'years'
  "exp" from emp where empno = 7902;
```

Note:
We are not allowed to use dml stmts in functions

Out:
If we want to return a function with more than one value then we use out parameter

→ write a pl/sql stored function for passing deptno as a parameter return dname, loc from dept table using out parameter.

```
Create or replace function f1( p_deptno number,
                    p_dname varchar2, p_loc varchar2)
    return varchar2
is
begin
select dname, loc into p_dname, p_loc from dept
    where deptno = p_deptno;
return p_dname;
    end;
```

Execution: begin
:c a f1(10) (:b, :c);
end;
print b c;

Execution: Variable a varchar2(10);
Variable b varchar2(10);
Variable c varchar2(10);

```
Sql > begin
      :a := f1(10, :b, :c);
      end;
```

Sql print b c;

16/9/2012

→ We can also develop our own user defined aggregate functions as same as predefined aggregate functn and also we can use aggregate functions in group by clause.

Ex:

Create or replace function f1 (p_deptno number) ~~varchar2~~)

return varchar2

is

Cursor c1 is select ename from emp

where deptno: p_deptno;

begin

for i in c1

loop

~~end~~;

a := a || ' ' || i.ename;

end loop;

end;

Execution

sql> select deptno, f1(deptno) from emp

group by deptno

Note: Oracle 10g introduced wm_Concat() aggregate function which returns multiple values group wise and also this function accepts all data type columns

Ex 1) select deptno, wm_Concat(ename) from emp

group by deptno;

Ex 2) select deptno, wm_Concat(hiredate) from emp

group by deptno

Job wise ename.

```
select job, wm_concat (ename) from emp
group by job
```

→ All stored functions information stored under user_stored user_source, user_procedures.
We can also drop function using

```
drop function functionname;
```

Triggers

→ trigger is also same as stored procedure, it is automatically invoked whenever dml operations performed on table or view.
There are two type of triggers supported by oracle.

1) Statement level triggers
2) Row level triggers

In stmt level trigger, trigger body is executed only once per dml stmt whereas in row level trigger, trigger body is executed for each row for dml stmts.

Syntax:

```
create or replace trigger triggername
before / after  insert / update / delete  on
tablename
[for each row]  → row level trigger
[when condition]
```

trigger timing ← before / after
insert / update / delete → trigger event
trigger specification

Trigger Body:

[declare]
→ Variable declarations, Cursors;
Begin
==
end;
/

Row Level Triggers :-

→ In row level triggers, trigger body is executed for each row for dml stmt thats why we are using 'for each row' clause in trigger specification.

→ And also data is internally stored in two rollback segment qualifiers they are new, old

→ These qualifiers are used either in trigger body or in trigger when clause in the trigger specification

→ when we are using these qualifiers in trigger body we must use ':' in front of the qualifiers name

Syntax: :old.Columnname
:new.Columnname

	insert	update	delete
:old	✓	✓	X
:new	X	✓	✓

write a pl/sql row level trigger on emp table whenever user inserts salaries, salary should be above 5000

Create or replace trigger t1

before insert on emp

for any kind of insertion we are before

~~for each row~~

insert for each row

begin

for if (:new.sal < 5000)

then

raise_application_error (-20,123, 'sal should be above 5000');

end if;

end;

SQL> insert into emp (empno, sal) values (1, 3000);

sal should be above 5000.

Write a pl/sql row level trigger on ~~row level~~ emp table not to delete records if emp sal below 3000

Create or replace trigger t1

before delete on emp

for each row

begin

if (:old.sal < 3000

then

raise_application_error (-20,123, 'donot delete records');

end if;

end;

→ write a pl/sql row level trigger on emp table whenever modifying salary of the employee always new sal more than the old sal

```
Create or replace trigger tr
before update on emp
for each row
begin
if :new.sal < :old.sal then
raise_application_error (20,123,'we cannot update the
                          record');
end if;
end;
```

→ write a pl/sql row level trigger using emp, dept tables implement on delete cascade concept. ~~Create table~~

```
Create or replace trigger tr
after delete on dept
for each row
begin
delete from emp where
deptno: :old.deptno;
end;
```

Execution

sql> delete from dept where deptno:10.

write a pl/sql row level trigger on dept table when user modifying deptno on dept table automatically those modifications affected in emp table

```
Create or replace trigger tr1
after update on dept
for each row
begin
update emp set
deptno = :new.deptno where
deptno = :old.deptno;
end;
```

Execution:

```
sql> update dept set deptno=1 where deptno=10;
sql> select * from dept;
```

18/9/2012

Auto Increment: If we want to generate primary key values automatically all database systems uses auto increment concept. But oracle if we want to implement this concept we use sequences, triggers i.e we are designing sequence in sql engine & that sequence in pl/sql trigger

```
sql> create sequence s1
     start with 1;
Sequence created
sql> Creat or replace trigger tr1
     before insert on test
     for each row
     begin
     select sal.nextval into
     :new.sno from dual;
     end;
```

```
sql> create table test (sno number(10)
     primary key, name
     varchar2(10));
table created.

sql> insert into test (name)
     values ('abc');
sql> insert into test (name)
     values ('xyz');
sql> select * from test;
```

*** Note: Oracle 11g introduced variable assignment concept when we are using sequences in pl/sql block. i.e "without using select into clause", "without using dual" table also we are using sequences in pl/sql blocks

syntax:
```
begin
variable name := sequence name.nextval;
end;
```

Ex: sql> Create or replace trigger tr1
before insert on tst
for each row
begin
:new.sno := s1.nextval;
end;
/

⇒ Instead of 'before' if we use after it displays an error msg

Ex: Create or replace trigger tr1
after insert on tst
for each row
begin
:new.sno := s1.nextval;
end;
/

Error: Cannot change new values for this trigger type

→ Before & After triggers : whenever we are submitting DML statements values are effected in database but in this case when we are creating trigger the values are effected in db based on the trigger timing in before trigger body is executed before the values are effected into db where as in after trigger trigger body is executed after dml statement values effected into db. Based on this mechanism developers choose before & after triggers.

→ If we want to restrict invalid data entry we must use before triggers. whenever one table values are effected based on another table values, then we use after trigger

→ whenever we are modifying the data using :new qualifier then we must use before trigger

Ex: write a pl/sql row level trigger whenever user inserting data into ename column automatically those values are converted into uppercase.

```
create or replace trigger tr1
before insert on emp
for each row
begin
:new.ename := upper(:new.ename);
end;
```

```
SQL> insert into emp(empno, ename) values(1, 'abc');
SQL> select * from emp;
```

Statement Level Triggers:

In this SLT, trigger body is executed only once for dml stmts. It does not contain :new, :old qualifiers. Generally SLT are used to define time component based conditions.

write a pl/sql stmt level trigger on emp table not to perform dml operations on saturday & sunday.

```
Create or replace trigger tr1
Before Insert or update or delete on emp

begin
if to_char (sysdate, 'DY') in ('SAT', 'SUN') then
raise_application_error ('we cannot perform dml operations
                          on SAT & SUN');

end if;
end;
```

→ we are not allowed to use when conditions in stmt level trigger.

write a pl/sql stmt level trigger on emp table not to perform dml operations on last day of the month

```
Create or replace trigger tr1
before insert or update or delete on emp

begin
if sysDate = last_day(sysdate) then
raise_application_error (-20345, 'we cannot perform
                          dml operations');
```

```
end if;
end;
/
```

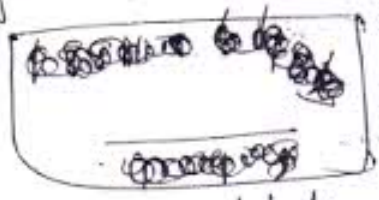
Note1:
```
if To_char(sysdate) = To_char(last_day(sysdate)) then
```

Note2:
```
if To_char(sysdate 'DY') = To_char(last_day(sysdate),
                                   'DY') then
```

20/9/2012

Trigger Predicate Clause:

These clauses are used either in start level of row level triggers. If we want to use multiple tables, multiple conditions in trigger body then only we use trigger predicate clause, they are



Inserting, updating, deleting clauses. These clauses are called as triggering events.

Syntax:
```
if inserting then
  starts;
Elsif updating then
  starts;
Elsif deleting then
  starts;
End if;
```

→ write a pl/sql stat level trigger on emp table not to perform any del operations in any days using trigger predicate clause

```
create or replace trigger two
before insert (or) update (or) delete on emp
begin
if inserting then
raise_application_error (-20123 'we cannot perform insertion');
elsif updating then
raise_application_error (-20123 'we cannot perform updation');
elsif deleting then
raise_application_error (-20123 ' cannot perform deletion');
end if;
end;
/
```

```
sql> create table test (msg varchar2(20));
sql> create or replace trigger two
     after insert (or) update (or) delete on emp
     declare
     z varchar2 (20);
     begin
     if inserting then
       z := 'rows inserted';
     elsif updating then
       z := 'rows updated';
     elsif deleting then
       z := 'rows deleted';
```

```
end if;
insert into test values (7);
end;
/
```

Execution

```
Sql> insert into emp (empno) values (3);
 Sql> insert into emp (empno) values (5);
Sql> update emp set sal = sal + 100 where deptno : 10;
   3 rows updated
Sql> delete from emp where empno in (3 , 4)
      3 rows deleted
Sql > update emp set sal : sal + 100;
  sql> select * from test;
       rows inserted
            ..
            ..
          updated
          deleted
```

→ write a pl/sql row level trigger on emp table whenever we are inserting a data into emp table, those rows are automatically stored in another table and also whenever we are modifying the data automatically old values are stored in another table and also whenever deleting data those rows automatically stored in another table using trigger predicate clauses

```
Sql> create table t1
      as select * from emp where 1 = 2;
Sql> Create table t2
     as select * from emp where 1 = 2;
```

SQL> Create table T3
as select * from emp where 1=2;

Create or replace trigger two
after insert (or) update (or) delete on emp
for each row
begin
if inserting then
insert into T1 values (:new.empno, :new.ename...);
elsif updating then
insert into T2 values (:old.empno, :old.ename...);
elsif deleting then
insert into T3 values (:old.empno, :old.ename...);
end if;
end;

Execution order of triggers

1) Before stat level
2) Before row level
3) After row level
4) After stat level.

Whenever we are using same level of triggers on same table we cannot control execution order of the triggers. To overcome this probl. oracle 11g introduced "follows clause" in triggers, follows clause used in trigger specification only. follows clause gives "guarantee execution order" explicitly.

Syntax: Create (or) replace trigger triggername
before | after | insert | update | delete on tablename
[for each row]
follows anotherr triggername
[declare]
Begin
end;

21/9/2012 Write a pl/sql row level trigger on test table whenever user inserting data into test table generate first columnname as sequence nextval no and also implement same level of trigger on same table.

This trigger get first column value & store it into second column

```
Sql> Create table test (col1 number(10), col2 number(10),
     cols date);
Sql> Create sequence s1 start with 5678;
Sql> Create or Replace trigger ts1

     before insert on test
     for each row
     begin
     select s1.nextval into :new.col1 from dual;
     dbms_output.put_line ('trigger1 fired');

     end;
     /

Sql> Create or Replace trigger ts2
     before insert on test
     for each row

     declare
     v_col2 varchar2(10);
     begin
     select
     reverse (to_char (:new.col1))
     into v_col2 from dual;
     :new.col1 := v_col2;
     dbms_output.put_line ('trigger2 fired');

     end;
     /
```

Execution

```
sql> insert into
     test (cols) values (sysdate);
     trigger 2 fired
     trigger 1 fired
sql> select * from test;
```

col1	col2	col3
5678		21-Sep-12

Solution (Oracle 11g)

```
Create or replace trigger ts2
before insert on test
for each row
follows ts1
declare
v_col2  varchar2(10);
begin
select
reverse (to_char (:new.col1));
  into v_col2 from dual;
  :new.col2 := v_col2;
dbms_output.put_line ('trigger 2 fired');
  end;
  /
```

O/P

col1	col2	col3
5678	8765	21-Sep-12

→ Calling Procedure into Trigger using Call statement.

using Call stmt we are explicitly calling a procedure into trigger

Syntax:

Create or replace trigger tg1
before / after insert/update / delete
on tablename
Call procedure name

Sql> select * from emp;

To have the total sum of sal from emp into some variable & getting update automatically with the changes

Sql> Create table test (tot number(10));

Sql> Create or replace procedure p1
is
v_sal number(10);
begin
delete from test;
select sum(sal) into v_sal from emp;
insert into test values (v_sal);
end;

Sql> set Create or replace trigger
tg5

after insert or update or delete on emp

Call p;

```
SQL> select * from test;
      tot
    14819
SQL> update emp set sal = sal - 100;
SQL> select * from test;
      tot
    14819
```

---- gets update automatically

After deleting the records write a pl/sql trigger on emp table whenever user deletes records from emp table automatically display remaining number of records number in bottom of delete stmt

```
SQL> Create or replace trigger ts1
     after delete on emp
     declare number(10);
     select count(*) from emp;
SQL> Create or replace trigger ts1
     after delete on emp
     declare
     t number(10);
     begin
```

Create or replace trigger ts1 after delete on emp

O/P

delete from emp where empno = 3;

15 rows

1 row deleted

if tot no: 0 records: 16

1 record deleted

remaining : 15

select count(*) ~~from emp~~ into z from emp;

dbms_output.put_line(z);

end;

/

Execution delete from emp where empno = 7902;

13

when no. of rows are affected we should use row level
but in the above app we have used stat trigger
The problem using row level trigger is, it displays an error
as ora-4091 : Scott.Emp is mutating

create or replace trigger

after delete on emp

for each row

declare

z number(10);

begin

select count(*) into z from emp.

dbms_output.put_line(z);

end;

/

row table is should which
leads to an error

it creates a trigger but when a value is passed like

SQL> delete from emp where empno=1;

error

Mutating : If a row level trigger based on a table, then trigger body cannot read data from same table and also we cannot perform dml operations on that table If we are trying this oracle server raises an error as

ORA-4091 : table is mutating

This error is called mutating error.
This table is called mutating table
Trigger is called mutating trigger

In stat level trigger, database server doesnot returns mutating error, because whenever we are performing dml operation -> using stat level trigger automatically transactional data permanently commited whereas in row level trigger transactional data not commited automatically, thats why without commited data reading from database using trigger automatically database server returns error.

-> To overcome this problem oracle server explicitly provides saving mechanism for the transactional data using autonomous transactions & using global variables

-> Always autonomous transactions avoids mutating errors but autonomous transaction returns previous results.

using autonomous transactions

```
create or replace trigger tss
after delete on emp
declare
 z number(10);
Pragma autonomous_transaction;
begin
select count(*) into z from emp;
dbms_output.put_line(z);
end;
/
```

```
sql> delete from emp where empno:1;
15 → expected result and 14 but it
     displays previous result.
```

22/9/12

ddl triggers

→ database administrators only create ddl triggers, on schema level
i.e these triggers are created
→ These triggers are also called as system triggers because administrator creates triggers on database level.

syntax: create or replace trigger triggername
before/after create/alter/drop/truncate/rename
- on username.schema

declare
→ variable declarations ;

begin
- - -
end;

write a pl/sql trigger on scott schema not to drop emp table

Create or replace trigger tr10
before drop on ~~schema~~ scott.schema
begin
if ora_dict_obj_name = 'EMP' and
if ora_dict_obj_type = 'TABLE' then
raise_application_error (-20543, 'we cannot drop emp table');
end if;
end;
/

o/p: drop table emp ;
ora-20543 - we cannot drop emp table

→ There are 12 types of triggers supported by oracle based on
before, statement, row, update, after, delete, insert

stmt ... > after | before → insert | update | delete

& also oracle supports instead of triggers on views
and also supports ddl triggers on schema level & system
triggers on database level

Enable / disable trigger

→ Enable / disable single trigger:

Syntax: alter ~~table~~ table
alter trigger triggername
enable / disable

→ Enable / disable all triggers in a table:

Syntax: alter table tablename
enable / disable all triggers;
space

Ex: alter table emp disable all triggers;

→ we can also drop trigger using
drop trigger triggername

→ All triggers information stored under user_triggers data dictionary.

→ Sql > desc user_triggers;

Packages

→ package is a database object which encapsulates variables, Constants, procedures, functions, types, cursors into single unit.

→ packages doesnot accept parameters, cannot be nested & also cannot be invoked directly

→ Generally packages are used to improve performance of the application, because when we are calling a packaged sub program first time automatically total package is loaded into memory area.

→ Whenever we are calling subsequent sub program calls oracle server executes those sub programs from mem. area. This process automatically reduces disk i/o

→ Packages contains two parts
  1.) Package Specification
  2.) Package Body.

→ When we are defining data within specification internally it behaves like a public data whereas package body data behaves like a private data i.e in package specification we declare sub programs whereas in package body we implement sub programs.

→ Syntax for package Specification:

```
Create [or replace] package packagename
is / As
→ global variable declarations, constants.
→ Cursor declarations;
→ procedure declarations;
→ function declarations;
→ types declarations;
end;
```

Syntax for Package body

```
Create or replace package
body packagename
Is/As
→ Procedure implementation;
→ function implementation;
end;
```

Invoking Packaged Subprograms

Syntax:

method1

```
sql> exec packagename.procedurename (actual parameters);
```

Syntax:

```
sql> select packagename.functionname(actual parameters);
```

Ex:

```
Create or replace package p1
is
procedure p1;
procedure p2;
end;
/
sql> Create or replace package
body p1
is
Procedure p1
is
```

```
begin
dbms_output.put_line ('first proc');
end p1;

Procedure p2
is
begin
dbms_output.put_line ('second proc');
end p2;
end;
/
```

Execution:

```
SQL> exec pn1.p2;
        second proc
SQL> exec pn1.p1;
        first proc.
```

*** Forward declaration

→ whenever we are calling procedure within another proc then we use forward declaration when we are calling local proc into global proc first we must implement local proc within package body before calling otherwise use forward declaratn in package body

Ex SQL> create or replace package pn1
is
procedure p1;
end;
/

```
Sql> Create or replace package body pm1
     is
     procedure P2;  → fawoard declaration
     procedure P1.

     is
     begin
     P2;
     end P1;
     Procedure P2
      is
     begin
      dbms_output.put_line ('local proc');

        end P2;
        end;

Sql> exec pm1.P1;
```

23/9/2012

Overloading Procedures

→ we can also implement overloading sub programs using packages
Overloading refers to same name with different parameters
overloading procedures contain same proc name & diff argum
-ents.

Ex: Create or replace package p2.

```
?is
procedure P1 (a number, b number);
procedure P1 (x number, y number);

end;
/
```

```
create or replace package body PZ1
  is
Procedure P1(a number, b number)
  is
c number(10);
begin
 c:= a+b;
 dbms_output.put_line (c);
 end P1;
 Procedure P1 (x number, y number)
 is
 z number(10);
 begin
 z:= x-y
 dbms_output.put_line (z);
 end P1;
 end;
 /
```

Execution.

```
Sql> exec PZ1.P1 (a => 7, b => 4);
       ||
Sql> exec PZ1.P1 (x => 6, y => 3);
```

when the overloading proc with same type of data types are used we should execute by using named notations

Types used in packages

→ In oracle we can also create our own user defined datatypes using type keyword.

Types are created in 2 step process

→ first we create type using appropriate syntax

then we create a variable of that type

→ Oracle server supports following types.

1.) pl/sql record

Collections ←
- *** 2.) Index by Table (or) pl/sql table (or) Associative Array.
- 3.) Nested Table
- 4.) Varray

*** 5.) Ref Cursor

1.) Pl/sql Record:

This is an user defined type which is used to represent different data types into single unit, it is also same as structures in c lang.

This is an user defined type, so we create in two step process

→ first we create type

→ create variable of that type.

Syntax: i) Type typename is Record (attribute : datatype(size)
attribute: datatype(size) ...);

ii) variable name Typename;

Ex: Create or replace package pkg
is
type t1 is record (a1 number(10), a2 varchar2(10),
a3 number(10));

procedure P1;

end;
/

package specification Created

Create or replace package body pkg

is
procedure P1
is
v_t t1;
begin
select empno, ename, sal into v_t from emp where
ename = 'KING';
dbms_output.put_line (v_t.a1 || ' ' || v_t.a2 || ' ' || v_t.a3);

end P1;
end;
/

Execution

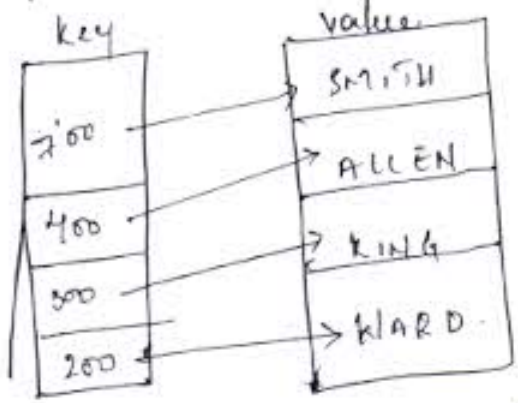
```
SQL> exec pk2.p1;
7839 KING 5600
```

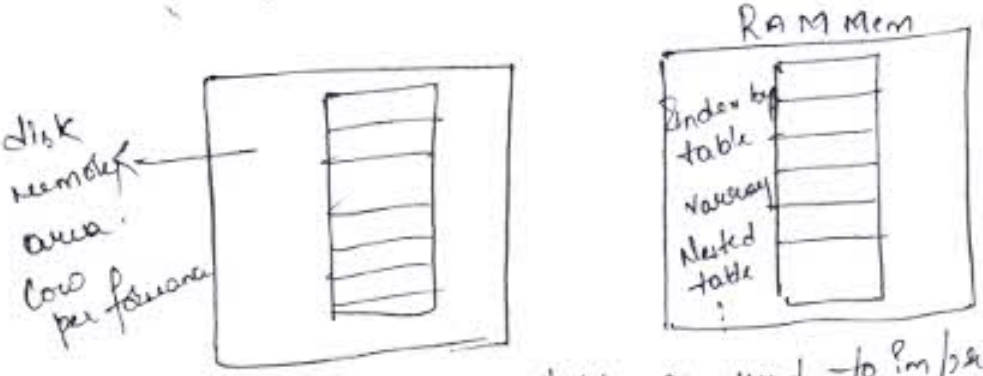
2) Index by Table :

→ These are user defined types which is used to store multiple data items. Basically this is an unconstraint table

→ Basically index by table has key value pairs i.e actual data stored in value field & indexes are stored in key field. These Indexes start with -ve to +ve numbers, these are not consecutive.

→ This key behaves like a PK key





→ Generally Index by tables are used to improve performance of applicatn because these are created in RAM mem area.

Here these tables are called as memory tables.

→ To improve performance of appl^n we define key as binary_integer datatype.

→ This is a user defined type, so we create in 2 step process.

→ we create type

→ we create a variable of that type.

Syntax: 1) Type Typename is table of datatype (size)

Index by binary_integer;

2) variable name Typename;

Ex: declare

type t1 is table of number(10)

Index by binary_integer;

v_t t1;

begin

v_t(1) := 10;

v_t(2) := 20;

v_t(3) := 30;

v_t(4) := 40;

dbms.output.put_line (v_t(3)); || 30

dbms_output.put_line (v_t.first); || 1

" (v_t.last); || 4

" (v_t.prior(3)); || 2

" (v_t.next(3)); || 4

```
dbms_output.put_line(v_t.count); ||4
v_t.delete(1,3);
dbms_output.put_line(v_t.count); ||1
v_t.delete;
dbms_output.put_line(v_t.count); ||0
end;
/
dbms_output.put_line(v_t(v_t.first)); ||10.
end;
/
```

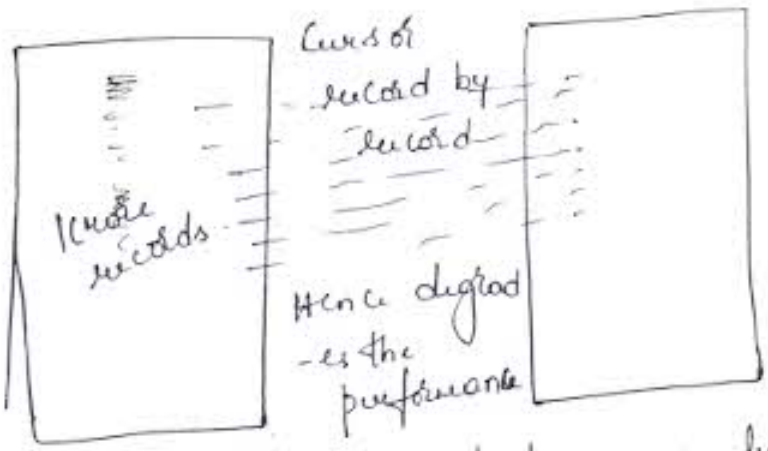
* Write a pl/sql prog to transfer all emp names from emp table & store it into index by table & also display content from index by table

```
declare
type t1 is table of varchar2(10)
index by binary_integer;
v_t t1;
Cursor c1 is select ename from emp;
n number(10) := 1;
begin
open c1;
loop
fetch c1 into v_t(n).
exit when c1%notfound;
```

storing into index by table.

```
    n := n+1;
    end loop;
    close c1;
for i in v_t.first .. v_t.last
loop
dbms_output.put_line(v_t(i));
end loop;
end;
/
```

for displaying the data



To overcome the above prob we use bulk collect.

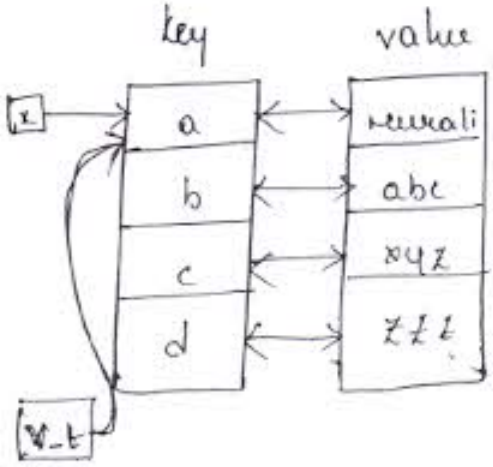
```
declare
type t1 is table of varchar2(10).
index by binary_integer;
v_t t1;
begin
select ename bulk collect into v_t from emp;
for i in v_t.first .. v_t.last
loop
dbms_output.put_line(v_t(i));
end loop;
end;
```

→ declare
type t1 is table of date
index by binary_integer;
v_t t1;
begin
for i in 1..10
loop
v_t(i) := sysdate+i;
end loop;
for i in v_t.first .. v_t.last
loop
dbms_output.put_line(v_t(i));
end loop;
end;

Write a pl/sql program to transfer all joining date of employees from emp table store it into index by table & display it from index by table.

declare
type t1 is table of date
index by binary_integer;
v_t t1;
begin
select hiredate bulk collect into v_t from emp;
for i in v_t.first .. v_t.last
loop

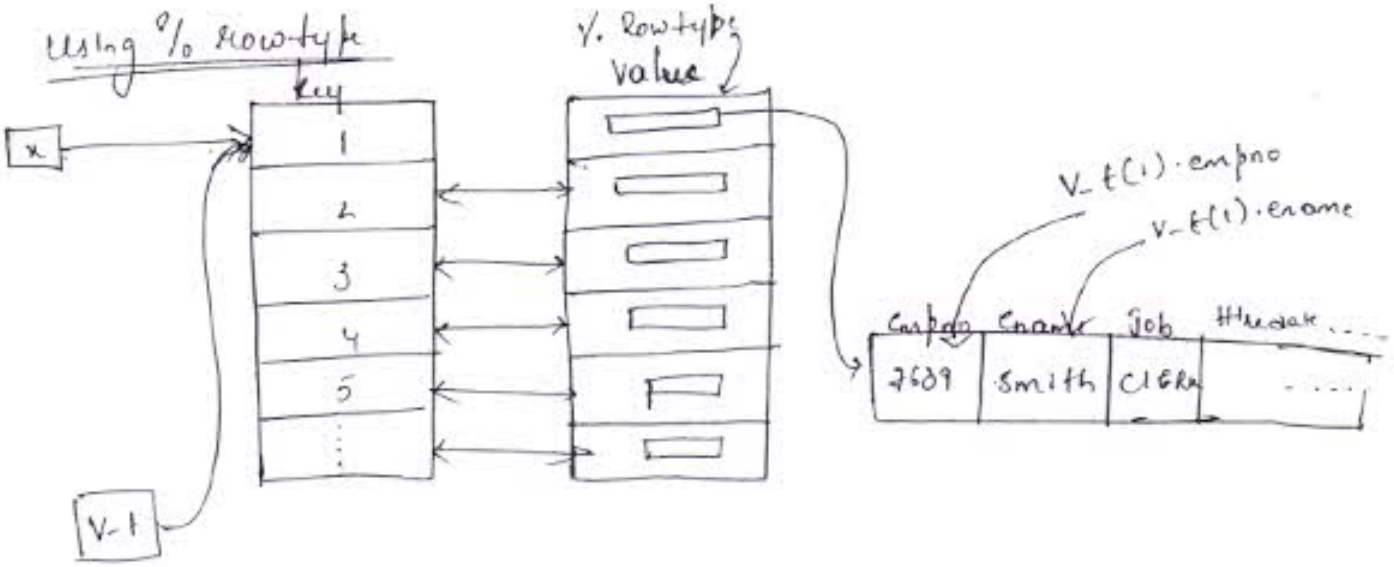
```
dbms_output.put_line(v_t(i));
end loop;
end;
```



as key & value pair
Contain dat varchar data hence
for loops are not used
as the loop by default takes integer values

Ex:

```
declare
type t1 is table of varchar2(10)
index by varchar2(10);
v_t t1;
x varchar2(10);
begin
v_t('a') := 'murali';
v_t('b') := 'abc';
v_t('c') := 'xyz';
v_t('d') := 'zzz';
x:='a';
loop
dbms_output.put_line(v_t(x));
x:= v_t.next(x);
exit when x is null;
end loop;
end;
```



declare
type t1 is table of emp%rowtype
index by binary_integer;

v_t t1;
x number(10);

begin
select * bulk collect into v_t from emp;

x := 1;

loop
dbms_output.put_line (v_t(x).empno || ' ' || v_t(x).ename
|| ' ' || v_t(x).hiredate);

x := v_t.next(x);

exit when x is null;

end loop;

end;
/

→ declare
type t1 is table of emp%rowtype
index by binary_integer;
v_t t1;
begin
select * bulk collect into v_t from emp;
for i in v_t.first .. v_t.last
loop
dbms_output.put_line(v_t(i).ename||' '||v_t(i).hiredate);
end loop;
end;

Return Result Sets :

→ if we want to return bulk of data from oracle database server into client app^n then we use following methods

i) using index by tables
ii) using ref cursors

→ when we are using these type of app^n we must implement server app^n using func^n & also execute using pl/sql client

i) Using index by tables
database server app^n

Ex: Create a replace package p3

is
type t1 is table of emp%rowtype

```
index by binary_integer;
 function f1 return t1;
  end;
  /
create or replace package body pkg
 is
 function f1 return t1
  is
  v_t  t1;
  begin
  select * bulk collect into v_t from emp;
   return v_t;
   end f1;
   end;
```

Execution

```
Sql> declare
      x pkg.t1;
      begin
      x:= pkg.f1;
      for i in x.first.. x.last
      loop
     dbms_output.put_line (x(i).ename || ' ' ||x(i).sal
                                || ' ' ||x(i).hiredate);
      end loop;
      end;
```

* Ref Cursor

→ This an user defined type used to process multiple records & also this is record by record process.

Generally static Cursors execute only single select start at a time for an active set area at Compile time whereas in Ref Cursor Oracle server executes no. of select starts dynamically for single active set area that why these Cursors are also called as dynamic cursors.

Oracle 7.2 Introduced Ref Cursors

→ Generally we are not allowed to pass static Cursor as parameters to sub program, To overcome this problem ANSI/ISO SQL Introduced Ref ~~Cursors~~ Cursors

These are user defined types, so these are passed as parameters to sub programs.

There are 2 types of Ref Cursors supported by Oracle

1) Strong Ref-Cursor.
2) Weak Ref-Cursor

Strong Ref-Cursor : is a Ref Cursor which has a return type while weak Ref Cursor doesnot have a return type

→ This is an user defined type so we create a two step process

→ we create a type
→ Create a variable of that type

→ Thats these cursors also called as Cursor variables

Syntax: i) Type typename is Ref Cursor
return record-type/data-type;

ii) Variable name Typename;
↙
strong Ref Cursor

weak Ref Cursor

Type Typename is Ref Cursor;

Variable Typename;

↙
weak Ref Cursor.

→ using open for stmts we execute select stmts.

Syntax: open Ref Cursor variable for select * from
tablename where condition

This stmt is used in executable section of plsql block.

Ex: declare
type t1 is Ref Cursor;
v_t t1;
? emp%rowtype;
begin
open v_t for select * from emp where sal > 2000;
loop
fetch v_t into ?;
exit when v_t%notfound;
dbms_output.put_line (i.ename || ' ' || i.sal);

```
end loop;
close v_t;
end;
```

→ Write a pl/sql program using ref cursor when user deptno : 10 display 10th dept details from emp table & also when user enter deptno: 10 display 20th dept details from dept table.

```
declare
type t1 is ref cursor;
v_t t1;
i emp%rowtype;
j dept%rowtype;
v_deptno number(10) := &deptno;
begin
if v_deptno = 10 then
open v_t for select * from emp where deptno=10;
loop
fetch v_t into i;
exit when v_t%notfound;
dbms_output.put_line(i.ename || ' ' || i.deptno);
end loop;
close v_t;
elsif v_deptno=20 then
open v_t for select * from dept where deptno=20;
loop
fetch v_t into j;
```

```
exit when v_t %. not found ;
dbms_output.put_line (i.deptno || ' ' || i.dname || ' ' ||
                                          i.loc);
end loop;
end
close v_t;
end if;
end;
/
```

Passing Ref Cursor as parameter to procedure.

```
Create or replace package pz4
is
type t1 is ref cursor return emp%rowtype;
type t2 is ref cursor return dept%rowtype;
procedure p1 (p_t1 out t1);
procedure p2 (p_t2 out t2);
end;
Create or replace package body pz4
is
procedure p1(p_t1 out t1)
is
begin
open p_t1 for select * from emp;
end p1;
procedure p2 (p_t1 out t2)
```

```
is
begin
open p_t for select * from dept;
end p2;
end;
/
```

Execution

```
SQL> Variable a refcursor;
sql> variable b refcursor;
sql> exec pkg4.p1(:a);
SQL> exec pkg4.p2(:b);
     SQL> print a b;
```

Note: Oracle 9i introduced sys_refcursor in place of weak refcursor

Syntax: Variable name sys_refcursor;

```
declare
v_t sys_refcursor;
i emp%rowtype;
begin
open v_t for select * from emp where sal > 2000;
loop
fetch v_t into i;
exit when v_t%not_found;
dbms_output.put_line(i.ename||' '||i.sal);
```

```
end loop;
close v_t;
end;
/
```

24/9/2012

Nested Tables, Varray:

Oracle 8.0 introduced nested table, varrays these are also userdefined types which are used to store multiple data items

But before storing actual data, we must initialise these types using Constructs

Here Constructorname is same as type name

→ Generally we are not allowed to store index by tables permanently into database

To overcome this problem oracle introduced nested tables, varrays to store permanently into database using sql

→ And also we cannot add & remove indexes in index by table.

whereas in nested table, varrays we can also add & remove indexes using extend(), trim() collection methods

→ Nested Table :

→ Same like index by table, this is an unconstraint table but indexes start with '1' & also these indexes are consecutive.

→ This is an user-defined type, so we create in two step process

Syntax : Type Typename is Table of datatype (size) ;

Variable name Typename := Typename ( ); → Constructor.

→ Ex: declare
type t is table of number(10);
v_t t1 := t1 ;
begin
v_t.extend (500); → The required num should be specified exactly otherwise it won't work
v_t (500) := 10; → If this statement is not used it raises an error as subscript out of bound.
dbms_output.put_line (v_t(500));
end;
/

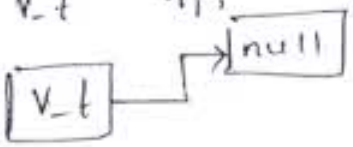
⇒ In nested table, varrays we can also store actual value through Constructor itself without using extend() Collection method

declare
type t1 is table of number(10);
v_t t1 := t1 (10, 20, 30, 40, 50);
begin

```
dbms_output.put_line (v_t.first );
                     (v_t.last );
                     (v_t.next(8) );
                     (v_t.prior(2));
                     (v_t.count);
for i in v_t.first .. v_t.last
  loop
  dbms_output.put_line (v_t(i));
  end loop;
  end;
  /
```

Case 1.) Type t_1 is table of number(10);

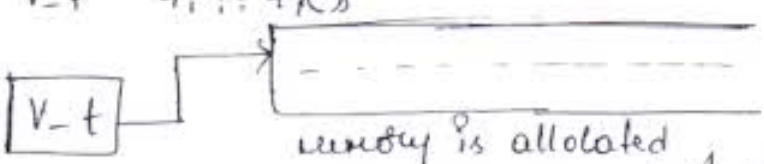
v_t t_1; → without constructor.



variable points to null.

Case 2.) Type t_1 is table of number(10);

v_t t_1 := t_1(); → Constructor.



memory is allocated when we use a constructor

```
declare
type t1 is of table of number(10);
  v_t1 t1;
  v_t2 t1 := t1();
```

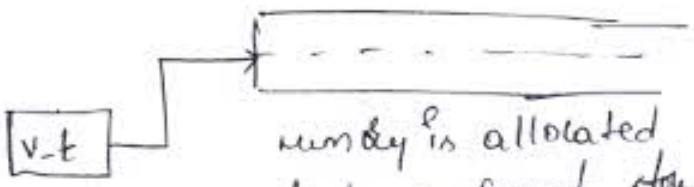
```
begin
if v_t1 is null then
dbms_output.put_line('v_t1 is null');
else
dbms_output.put_line('v_t1 is not null');
end if;
if v_t2 is null then
dbms_output.put_line('v_t2 is null');
else
dbms_output.put_line('v_t2 is not null');
end if;
end;
/
```

Output :

```
v_t1 is null
v_t2 is not null
```

Case 3 :

```
declare
type T1 is table of number(10);
v_t t1 := t1();        --> Constructor
begin
v_t(1) := 10;
dbms_output.put_line(v_t(1));
end;
```



memory is allocated
but we cannot store the value
we can store it by using extend()

o/p : error

Case 4:

```
declare
  type t1 is table of number(10);
  v_t t1 := t1();
begin
  v_t.extend;
  v_t(1) := 10;
  dbms_output.put_line (v_t(1));
end;
/
```

o/p : 10

write a pl/sql program to transfer all employee names from emp table & store it into nested table & also display content from nested table

```
declare
type t1 is table of varchar2(10);
  v_t t1 := t1();
  Cursor c1 is select ename from emp;
  n number(10) := 1;
begin
for i in c1
  loop
```

```
v_t.extend();
 v_t(n) := i.ename;
  n:=n+1;
  end loop;
for i in v_t.first .. v_t.last
 loop
dbms_output.put_line (v_t(i));
 end loop;
  end;
  /
```

(b)

```
declare
type t1 is table of varchar2(10);
v_t t1:= t1();
begin
select ename bulk collect into v_t from emp;
  for i in v_t.first .. v_t.last
   loop
dbms_output.put_line (v_t(i));
 end loop;
  end;
  /
```

Varray :

Oracle 8.0 introduced varray. It is user defined type used to store similar data-type, but this is a constraint table, it stores upto 2 GB data.

Same like nested table we must use Constructor before actual data stored into this user defined type

Syntax : Type Typename is Varray (max size) of data type (size);

Variable name Typename := typename ( );
↳ Constructor

Ex declare
type t1 is varray (10) of varchar2 (10);
v_t t1 := t1 ('a', 'b', 'c', 'd');
i boolean;
begin
dbms_output.put_line (v_t.limit);
(v_t.count);
(v_t.first);
(v_t.last);
(v_t.prior(3));
(v_t.next(2));
for i in v_t.first .. v_t.last
loop
dbms_output.put_line (v_t(i));

```
end loop;
z: v_t.exists(3);
if z: true then
dbms_output.put_line('Index 3 exists with an element'
                     || ' ' || v_t(3));
else
dbms_output.put_line('Index 3 does not exis');
end if;
v_t.extend;
dbms_output.put_line(v_t.count);
v_t(5):='e';
v_t.extend(2);
v_t(6):='f';
v_t(7):='g';
v_t.extend(3,4);
v_t.trim(5);
for i in v_t.first.. v_t.last
loop
dbms_output.put_line(v_t(i));
end loop;
v_t.delete;
dbms_output.put_line(v_t.count);
end;
/
```